MASTER IN COMPUTER SCIENCE

Calculus of Communicating Systems: A web based tool in Scala

Gillet, Jean-François; Willame, Danwel

Award date:
2017

Awarding institution:
University of Namur

Université de Namur
Faculty of Computer Science
Academic Year 2016–2017

Calculus of Communicating Systems:

A web based tool in Scala

Danwel Willame

Jean-François Gillet

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Jean-Marie Jacquet

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université de Namur

Calculus of Communicating Systems: A web based tool in Scala

Abstract

This master-degree thesis presents the design of a web-based tool for modeling and verifying concurrent processes. These are expressed in the process algebra of Robin Milner: *Calculus of Communicating Systems* (CCS). The tool allows comparison of processes by means of both simulation and strong bisimulation. The tool also allows to display CCS equations in the form of graphs.

Other similar tools were previously developed, but to the candidates' knowledge none were developed in Scala, a programming language that fuses functional and object-oriented paradigms. This thesis describes the functions of the tool, as well as its implementation, while emphasizing the expressiveness provided by Scala.

Ce mémoire traite du développement d'un outil web permettant de modéliser et de vérifier des processus concurrents. Ceux-ci sont exprimés dans l'algèbre des processus de Robin Milner, le *Calculus of Communicating Systems* (CCS). L'outil permet de comparer des processus grâce à la simulation et à la bisimulation forte. Il permet également de visualiser les équations CCS sous forme de graphes.

Des outils similaires ont été développés précédemment, mais à la connaissance des auteurs, aucun d'entre eux n'a été implémenté en Scala, un langage de programmation qui fusionne les paradigmes fonctionnel et orienté objet. Ce travail décrit les fonctionnalités de cet outil ainsi que leurs implémentations, tout en mettant l'accent sur l'expressivité apportée par Scala.

Contents

List of Figures

Acknowledgements

First, we express sincere gratitude to our supervisor, Prof. Jean-Marie Jacquet, for his continuous support during our work in developing this master's thesis, as well as his patience, motivation, and immense knowledge shared with us. We could not imagine a better advisor for this effort.

We also gratefully acknowledge Dr. Bruce G. Weniger, Adrien Houdart and Ludovic Bukens for their very valuable comments and insights on this work and this resulting thesis.

Finally, we take this opportunity to express gratitude to our families and significant others for their unceasing encouragements, support, and attention.

We live on an island surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance.

- John Archibald Wheeler

1 Introduction

Recent trends in computing have affected how computer scientists build software. The growing adoption of cloud computing, web services, and distributed systems, on the one hand, and the explosion of embedded systems on the other hand, have made concurrency as a prime player in software development [8, 20, 21]. However, the writing of such concurrent software can be laborious, even for relatively minor applications.

In his article on embedded software market trends, Gavhane Sagar[20] explains that the automotive industry provides a growing demand for such systems, and plays a key role in market growth for embedded software. Many embedded systems are safety-critical, such as: anti-lock brake system (ABS), electronic brake distribution (EBD), and electronic stability program (ESP). One can recall famous examples of bugs in embedded systems, as in the explosion of the unmanned Ariane 5 rocket in 1996 or the loss of the Mars Climate Orbiters in 1999 [13]. Similarly, a faulty component in large distributed systems can be also very damaging.

1.1 Transformational vs Reactive Systems

With such concerns in mind, formal verification is of great interest. Although software testing can help to find bugs, it cannot prove their absence. The standard view of computing considers systems as transformational black-boxes that transform input into output.

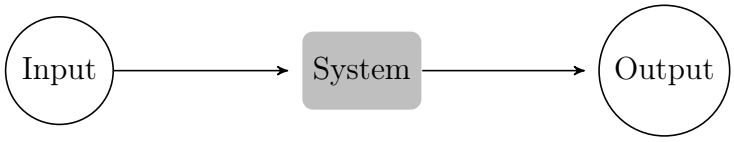


Figure 1.1: A transformational system

Therefore, algorithmic problems are specified by preconditions and postconditions i.e, what are their legal inputs and expected output [1, p. 3] [13, p. 7]. In non concurrent settings, Hoare logic is used to reason about the correctness of algorithms [9]. This logic describes how a system transforms an input step by step. At the end of the process, if the expected final state is reached the system is considered proved correct. The central feature of this logic is the Hoare triple:

$$\{P\}\ C\ \{Q\}$$

where P is the precondition, Q is the postcondition and C is the command. In other words, if P is satisfied, then the execution of the command C leads to Q. Partial functions can lead to non-terminating programs. A partial function is a function whose behavior is not defined for some inputs. The function will therefore never produce an output. In this view of computing, the non-termination is highly undesirable.

On the contrary, in systems such as web services, control programs or embedded systems, the termination reflects a deadlock state and is generally a bad system state: such systems must always be ready for interactions. Therefore, their behaviors cannot be described as a terminating function with an input and an output.

Reactive systems can be seen as systems running in parallel with their environment, reacting to their environment and exchanging interactions with it. The interaction becomes the basic unit of computation and systems can be specified as a triple $(Event, Condition, Action)$ where the sequence of interactions provides the computation [13, p. 7].

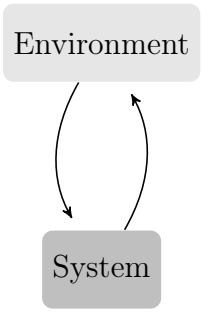


Figure 1.2: A reactive system

1.2 Process theory

Having defined that distributed and embedded systems are reactive systems and that their verification is of great importance, we now propose an adequate abstract model to define and verify them. Process theory provides a set of models and techniques to describe the behavior of parallel processes together with facilities to identify events such deadlock, livelock, starvation... The key elements of process theory, as stated by Luca Aceto et al. [1, p. 5], are:

- Logic;
- Process algebra;
- Labeled transition systems;
- Structural operational semantics.

The process algebra presented in this work is the Calculus of Communicating System (CCS) introduced by Robin Milner in 1980 [15]. Furthermore, we will introduce notions of behavioral equivalence to compare processes. These equivalences will allow us to verify if a process correctly implements its specification.

1.3 Development of a CCS workbench in Scala

To fulfill formal verification on reactive systems, we propose the development of a web-based tool for modeling and verifying concurrent processes expressed in Milner's Calculus of Communicating Systems. This tool will offer us a graphical view for displaying CCS formulae and for reasoning about them. It will also offer a range of equivalence notions to compare CCS processes.

Other tools were previously developed for CCS [4] but to our knowledge none were developed in Scala.

Scala is a programming language designed and developed in the programming methods laboratory at EPFL since 2001 [18]. "Scala fuses object-oriented and functional programming in a statically typed programming language." [17, p. 1]

Our work will try to emphasize the expressiveness offered in Scala by the union of functional and oriented object paradigms, and evidence that it provides a major means for the development of such a platform.

2

Calculus of Communicating Systems

The calculus of communicating systems (CCS) was introduced by Robin Milner in 1982 [15]. This language describes concurrent systems and provides us with an abstraction for them.

The benefits of such a formal algebra are manifold. With it, one can write a set of equations that depicts the specification for a system, and extract another set of equations which describes its actual implementation. Then, it is possible to verify whether the implementation satisfies its specification through equivalence checks[1]. Other correction properties can be checked as well on equations such as the absence of *deadlock*, *starvation* or *livelock* [1, p. 6-8].

Milner's primary observation was that concurrent systems have an algebraic structure. Indeed, one can build two processes A and B and then combine them with an operator to create a new process C. The behavior of C depends on both A and B, as well as the operation used to combine them.The underlying structure of such a collection of operations to build new processes from existing ones, is algebraic in the first sense [1, p. 7-8].

The following introduces and explains the grammar and syntax of CCS, using informal examples for a general idea of the language, and then its formal specifications.

[1]To be detailed in the chapter Behavioral Equivalences

2.1 CCS by example

Let's suppose you need to visit with a doctor. First, you may have to contact a staff worker in an administrative department. Then, as hospitals are usually crowded places and waiting queues can be long, a machine may provide you with a ticket, so that your arrival time corresponds fairly with the interval you must wait. The ticket reflects your position in the waiting queue.

Figure 2.1: Ticket machine

Figure 2.1 schematizes a ticket machine. It can be seen as a black box with two communication channels, one for the input *push* and the other for the output $\overline{ticket}$. From this model one can only extract static information. To describe its structure and behavior, one needs a CCS program. What follows is how to write one.

The most basic one is:

$$\emptyset \text{ or } Nil \tag{2.1}$$

This process accomplishes nothing. It does not make any further computation and thus depicts a deadlock state. To elaborate more complex ones, one has to use a constructor. The action prefixing is the most straightforward one, as illustrated here:

$$IE \stackrel{def}{=} DownloadChrome.0 \tag{2.2}$$

Intuitively, Internet Explorer (IE) is a process that dies once one has downloaded chrome. The above example illustrates how, CCS offers the possibility to name new processes composed from other processes. This allows reusability and composition. Such definitions can of course be recursive.

Now, we have enough understanding to define our ticket machine as

$$TM \stackrel{def}{=} push.\overline{ticket}.TM \tag{2.3}$$

The behavior of our device is thus : first an input action *push* is done, then an output action $\overline{ticket}$ is performed, and finally the process returns to its initial state of TM.

This system is rather simple. What about the need for two distinct waiting queues : one for consultations and another for hospitalizations ? The patient must choose between consultations and hospitalization to get a ticket for the correct queue. This is made possible by a choice operator $+$, as in

$$TM1 \stackrel{def}{=} (hospitalization.\overline{hTicket}.TM1 + consultation.\overline{cTicket}.TM1) \quad (2.4)$$

To improve the system one might add a sleep mode by appending a sleep action just after handing out a ticket. If so, then the system also requires a wake-up action in order to return to active mode. Such action is added at the beginning of the equation :

$$\begin{aligned} TM2 \stackrel{def}{=} wakeup.(&hospitalization.\overline{hTicket}.sleep.TM2 \\ &+ consultation.\overline{cTicket}.sleep.TM2) \end{aligned} \quad (2.5)$$

One can observe that the *wakeup* action, common to both choices, has been put out of the parenthesis. But can we distribute *wakeup* and have the same machine? Which system would you prefer if you have an appointment with a doctor for a consultation?

$$\begin{aligned} TM3 \stackrel{def}{=}\ & wakeup.hospitalization.\overline{hTicket}.sleep.TM3 \\ & + \\ & wakeup.consultation.\overline{cTicket}.sleep.TM3 \end{aligned} \quad (2.6)$$

In the second program you can push on *wakeup* and be in the first branch of the choice, which ends up with a hospitalization ticket. In the prior equation once you have pushed on *wakeup* both choices still remain available.

One can notice that the behavior of the equation $TM3$ can be simulated by the equation $TM2$ but the opposite is not true. This behavior will be detailed in Chapter 5 *Behavioral Equivalences*.

For the next step we need a patient to interact with our ticket device. Let us model him or her as follows:

$$P \stackrel{def}{=} \overline{push}.ticket.\overline{money}.P$$

Now, we can write a system with these two processes running in parallel. To do so, we use the parallel composition operator $|$.

$$H = TM|P$$

TM and P may communicate via their complementary ports although such communication is optional. Figure 2.2 illustrates the potential communication between them. The money port can be used by other processes in the environment of P, for example, the hospital accounting department process.

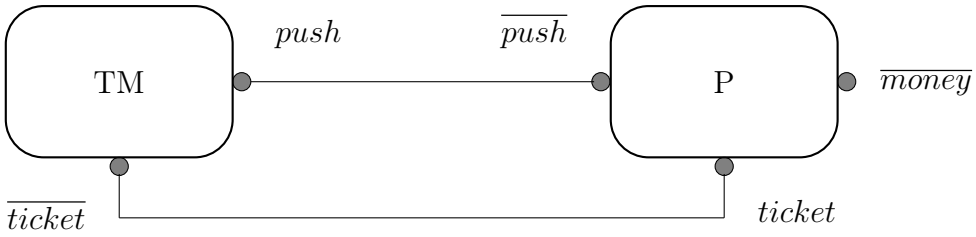


Figure 2.2: H=TM|P

Note that TM and P can use their ports to communicate with other reactive systems in their neighborhood. The ticket machine is able to communicate with other patients, as illustrated in Figure 2.3.

CCS offers an operator called restriction \. This operator makes the restricted channels inaccessible from the outside world. Doing so, our patient P can take an advantage over the others by making the ticket device only accessible by him as showed in the Figure 2.4.

$$PH \stackrel{def}{=} (TM|P)\backslash push \backslash ticket$$

The last CCS operator is for the renaming. The following examples introduce this operation:

$$SM \stackrel{def}{=} coin.\overline{soda}.SM$$

$$ChM \stackrel{def}{=} coin.\overline{chocolat}.ChM$$

$$CM \stackrel{def}{=} coin.\overline{coffee}.CM$$

These CCS programs describe three distributor machines available in the waiting room. The first gives you soda, the second chocolate and the third coffee. An astute reader can notice that the underlying behavior can be generalized as follows:

$$VM \stackrel{def}{=} coin.\overline{item}.VM$$

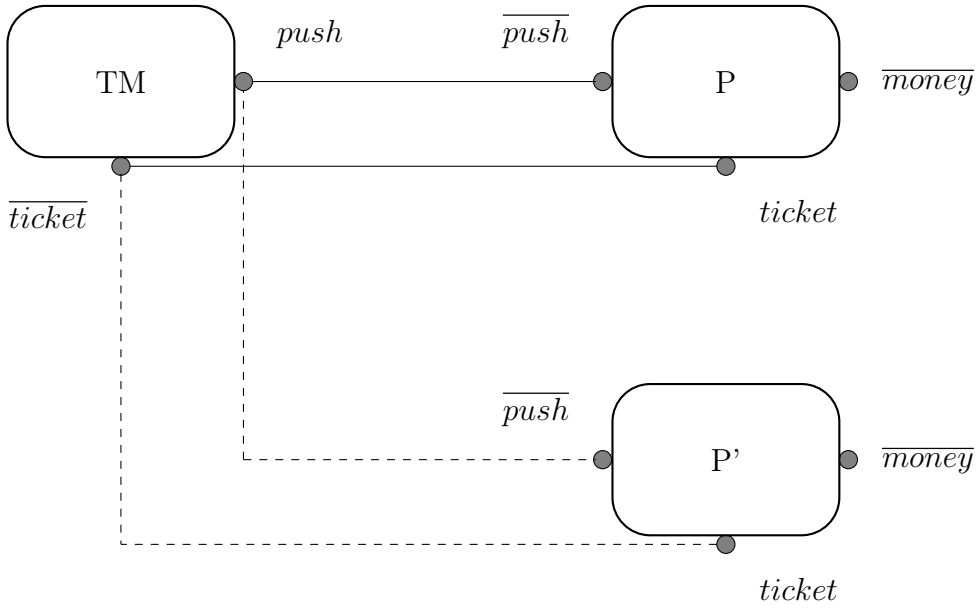


Figure 2.3: H= TM | P | P'

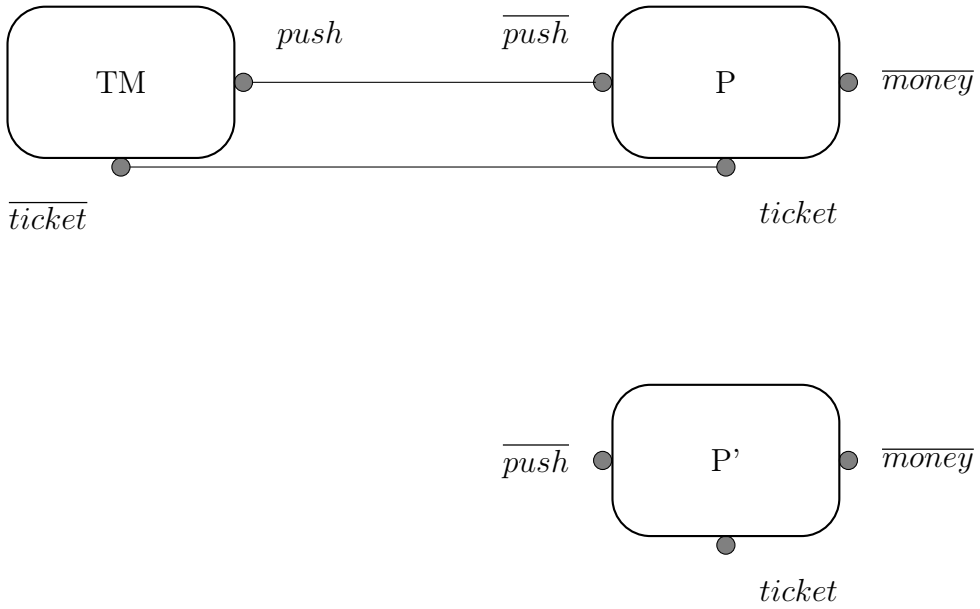


Figure 2.4: PH = (TM | P)\push \ticket

Upon helpful renaming of SM, this becomes:

$$SM \stackrel{def}{=} VM[soda/item]$$

2.2 Labeled Transition Systems – LTS

CCS programs can also be seen as finite automata [1, p. 18]. The key idea behind this view is that the CCS processes jump from states to states by performing actions.

The formal link between CCS and LTS will be given futher below. To highlight this idea, let's take process P and apply successive actions.

$$P \stackrel{def}{=} \overline{push}.ticket.\overline{money}.P$$

After performing $\overline{push}$ the system becomes as follows:

$$P1 \stackrel{def}{=} ticket.\overline{money}.P$$

The system can now receive a ticket and jump to the next state.

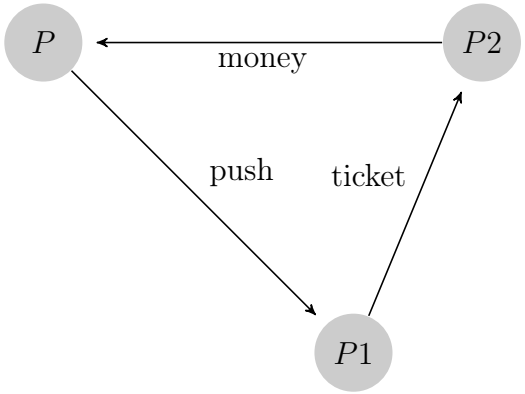
$$P2 \stackrel{def}{=} \overline{money}.P$$

Finally, once the patient has paid money, the systems goes back to the state P. The processes change their states through transitions. Those transitions are labeled by the action name that triggered them.

$$P \stackrel{\overline{push}}{\longrightarrow} P1$$

The operational semantic of our patient is given by the following transitions:

$$P \stackrel{\overline{push}}{\longrightarrow} P1 \stackrel{ticket}{\longrightarrow} P2 \stackrel{\overline{money}}{\longrightarrow} P$$



The set of states for our ticket machine is given by:

$$TM \stackrel{def}{=} push.\overline{ticket}.TM$$

$$TM1 \stackrel{def}{=} \overline{ticket}.TM$$

One can notice that patient P wishes to receive a ticket when he or she is in the state $P1$ and our ticket machine TM is able to deliver it when in the state $TM1$. A similar observation can be made about push.

In CCS, inter-process communications are performed via a handshake to avoid a third process to join the two synchronized processes [1, p. 15,16].

To do this, one uses the restriction operator to privatize the actions *push* and *ticket*. The state transition is therefore unobservable from the outside world. The process is going to change its state through an unobservable transition called τ. Figure 2.5 exhibits the τ transition for the process PH.

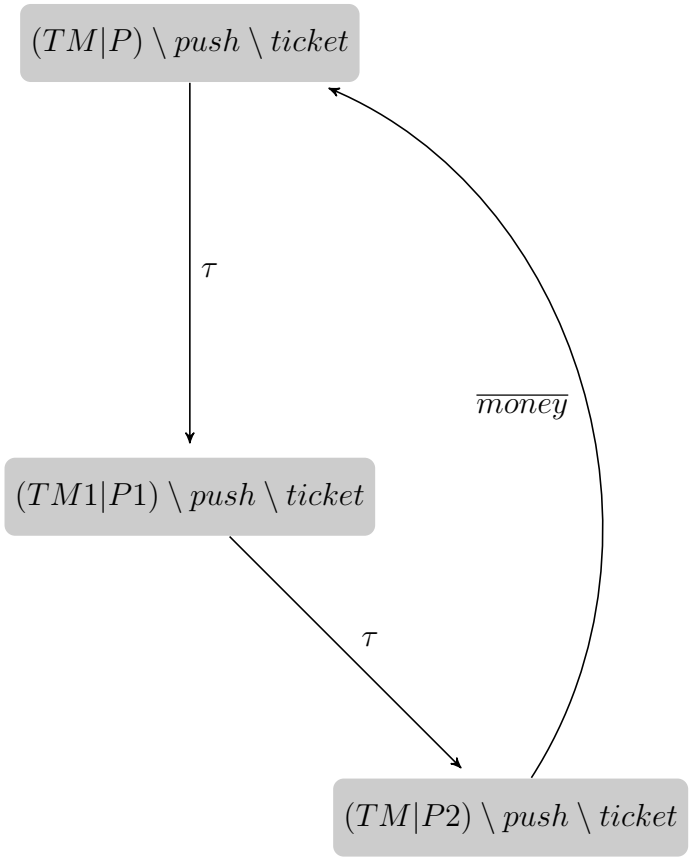


Figure 2.5: PH = (TM | P)\ push \ ticket

The following process exhibits the same behavior as PH, but with a different level of abstraction.

$$Spec \stackrel{def}{=} \overline{money}.Spec \tag{2.7}$$

If one considers PH as black-box, an external observer cannot see all the τ transitions triggered inside PH. For such an observer, PH and $Spec$ are indistinguishable. Therefore, one can say that $Spec$ is the specification of PH in the sense that all transitions performed by PH are authorized by Spec.

2.3 CCS, formally

Let us now turn to formal definitions for CCS, whose syntax complies with the following definitions:

- Let A be an infinite countable set of channel's names;
- $\overline{A}_{=}\{\overline{a} \mid a \in A\}$, is the complementary set of names;
- $L = A \bigcup \overline{A}$, is the set of labels;
- $Act = L \bigcup \{\tau\}$, the set of actions;
- K, is an infinite countable set of process names.

The set P of valid expressions of CCS is given by the following grammar:

$$P ::= k \mid \alpha.P \mid P + P \mid \sum_{i \in I} P_i \mid (P|P) \mid P[f] \mid P \backslash L$$

in which,

- P and the P_i 's are valid expressions;
- k is a process in K;
- α is an action in Act;
- I is an index set;
- $f : Act \rightarrow Act$ is a relabeling function satisfying the following constraints:

$$f(\tau) = \tau$$

$$f(\overline{a}) = \overline{f(a)} \text{ for all } a \in L,$$

- L is the set of labels;

As one may guess, $\sum_{i \in I} P_i$ is the generalization of the choice $P + P$ between two expressions:

$$\sum\nolimits_{i \in I} P_i = P_1 + P_2 + ... + P_n$$

By convention, if $I = \emptyset$ then $\sum_{i \in I} P_i$ is interpreted as 0 (zero).

As we have seen before, the operational semantics of CCS can be expressed by a finite automaton. This automaton is called a Labeled Transition System [LTS] in the concurrency theory.

An LTS is a triple of the form ($Proc$, Act, $\{\overset{\alpha}{\rightarrow} | \, \alpha \in Act\}$)

- $Proc$ is a set of states ranged over by s;
- Act is a set of actions ranged over by α;
- $a \rightarrow \subseteq Proc \times Proc$ is a transition relation, for each $\alpha \in Act$. A more intuitive notation exists: $s \rightarrow s'$ in place of $(s, s') \in \rightarrow$

Given this definition the LTS formal specification of PH is:

$$Proc = \{(TM|P)\backslash push\backslash ticket,\ (TM1|P1)\backslash push\backslash ticket,\ (TM2|P)\backslash push\backslash ticket\}$$

$$Act = \{money, \tau\}$$

$$\overset{money}{\longrightarrow} = \{(TM2|P)\ \backslash push\backslash ticket,\ (TM|P)\backslash push\backslash ticket)\}$$

$$\overset{\tau}{\longrightarrow} = \{(TM|P)\ \backslash push\backslash ticket,\ (TM1|P1)\backslash push\backslash ticket),\ (TM2|P)\ \backslash push\backslash ticket\}$$

An LTS transition is valid iff it complies with the Structural Operational Semantics rules (SOS). Each valid step that a CCS program can perform is given by the following rules :

Action. The first rule describes how actions are performed :

$$\textsc{Act} : \ \frac{}{\alpha.P \overset{\alpha}{\longrightarrow} P} \tag{2.8}$$

Act has no premises. This axiom means that each process of the form $\alpha.P$ is able to perform a transition α and then be in state P.

Procedure. The second rule defines how procedures are formed :

$$\textsc{Cst}: \ \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \ where \ K \stackrel{def}{=} P \tag{2.9}$$

For Cst to establish that the constant K can afford the transition α leading to P', one has to prove first that P itself affords the same transition, and finally that K equals P.

Choice. The third rule describes the choice transitions:

$$Sum_j: \ \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \ where \ j \in I \tag{2.10}$$

Sum describes the behavior of the choice operator $+$. If one of the processes contained in the choice affords a transition α that leads to the state P'_j, then the entire choice conducts to the state P'.

Parallelism. The following three rules depict parallelism :

$$\textsc{Com1}: \ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \tag{2.11}$$

$$\textsc{Com2}: \ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \tag{2.12}$$

$$\textsc{Com3}: \ \frac{P \xrightarrow{a} P' \ Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \tag{2.13}$$

Com1, Com2 and Com3 characterize the legal transitions for processes running concurrently. Com1 allows us to take the transition from the left side of the equation and Com2 from the right side. The rule Com3 can only be performed with complementary actions and will result in a τ transition.

Let's explain those rules with an example. Let's imagine we have the process $R = P \mid Q$ where $P = a.b.0$ and $Q = a.\overline{b}.0$. This will result in the following set of states:

$$\begin{cases} 1 & = & a.b.0 \mid a.\overline{b}.0 \\ 2 & = & b.0 \mid a.\overline{b}.0 \\ 3 & = & 0 \mid a.\overline{b}.0 \\ 4 & = & a.b.0 \mid \overline{b}.0 \\ 5 & = & a.b.0 \mid 0 \\ 6 & = & b.0 \mid 0 \\ 7 & = & 0 \mid \overline{b}.0 \\ 8 & = & b.0 \mid \overline{b}.0 \\ 9 & = & 0 \mid 0 \end{cases}$$

Figure 2.6 shows all the possible transitions for $R = P \mid Q$

Renaming. The next rule defines the renaming :

$$\textsc{Rel} : \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \tag{2.14}$$

If process P leads to P' by a transition α, and if f renames α to $f(\alpha)$, then the renamed process $P[f]$ reaches $P'[f]$ by rule Rel.

Restriction. The last rule defines the restriction :

$$\textsc{Res} : \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \; where \; \alpha, \overline{\alpha} \notin L \tag{2.15}$$

If a transition α goes from P to P', and if *alpha* and $\overline{\alpha}$ do not belong to the set of restricted actions, then the transition α for the process P is legal.

2.4 Passing by values

Even if not necessary from a theoretical point of view [16, p.53-56] an extension of the CCS language can be convenient to exchange data between processes. For the sake of simplicity, in the following rules the only data type to be used will be a set of natural numbers.

To do so, the prefixing rules for inputs become :

$$\frac{}{a(x).P \xrightarrow{a(n)} P[n/x]} \; for \, n \geq 0,$$

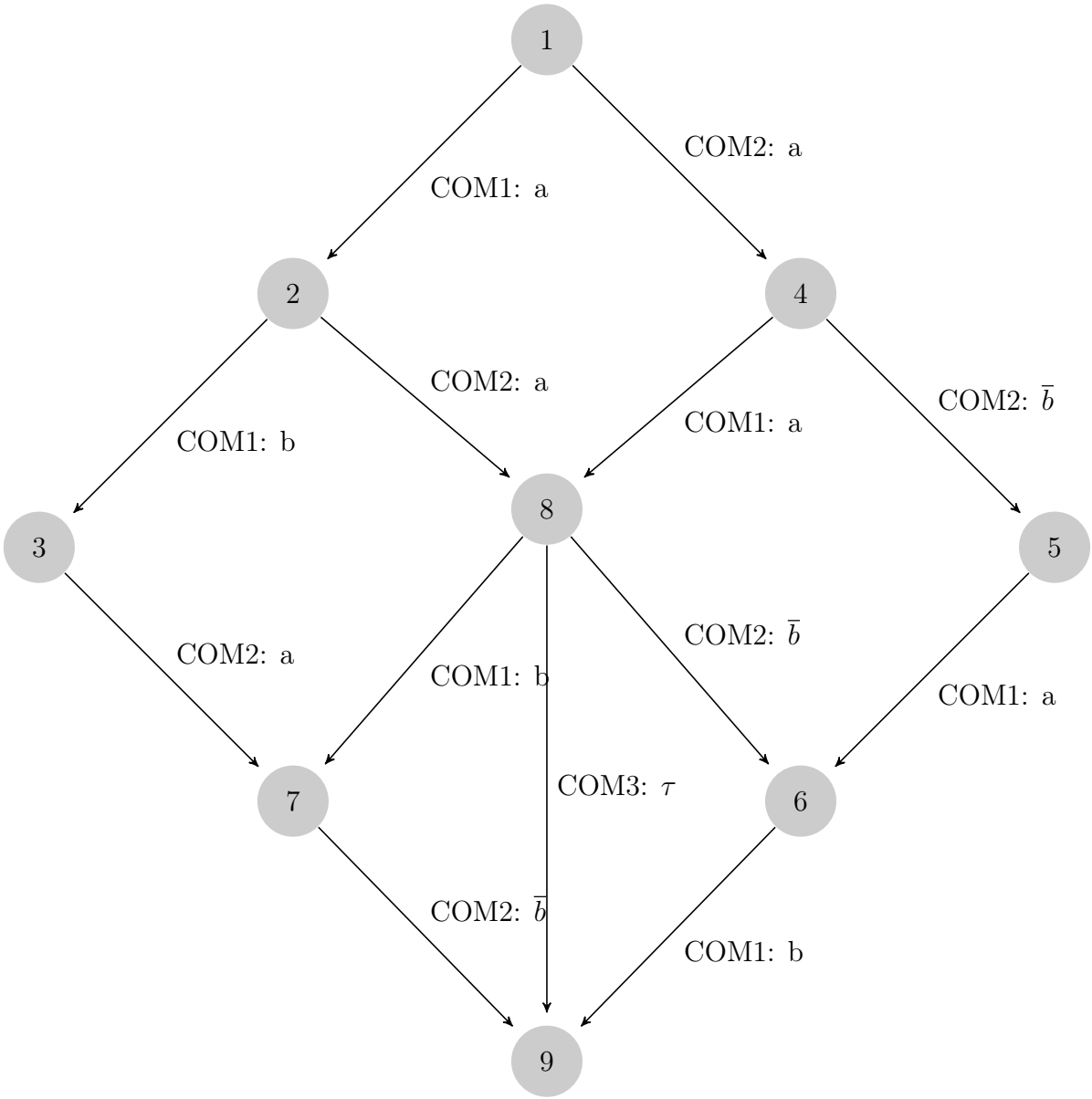


Figure 2.6: SOS transitions for $R = P \mid Q$

If there is parametrized input action a, then $P[n/x]$ replaces each free occurrence of the variable x by n. Therefore the input leads to the process $P[n]$.

For outputs:

$$\frac{}{\overline{a}(e).P \overset{\overline{a}(n)}{\rightarrow} P} \text{ where } n \text{ is the outcome of evaluating } e.$$

The processes can be parametrized as well. The operational semantic is given by:

$$\frac{P[v_1/x_1, ..., x_n/v_n] \xrightarrow{\alpha} P'}{A(e_1, ..., e_n) \xrightarrow{\alpha} P'} \; where \; A(x_1, ..., x_n) \overset{def}{=} P$$

- $x_1, ..., x_n$ are distinct values;
- $n \geq 0$;
- A is the process name;
- each e_i has for value v_i;

If there is a transition from $P(x) \xrightarrow{\alpha} P'$ and the constants $A(1, 2, ..., n)$ are defined as P, then $A(e_i)$ leads to P'.

Since CCS now manipulates values, it can be useful to have a conditional operator. Let's first give an example of a conditional operator semantic that will be used in the next example:

$$\frac{P \overset{\alpha}{\rightarrow} P'}{if \; bexp \; then \; P \; else \; Q \overset{\alpha}{\rightarrow} P'} \text{ if the boolean expression } bexp \text{ is } true$$

This rule maintains that if the evaluation of the boolean expression is true, then P does not perform a transition α and remains in the same state.

$$\frac{Q \overset{\alpha}{\rightarrow} Q'}{if \; bexp \; then \; P \; else \; Q \overset{\alpha}{\rightarrow} Q'} \text{ if the boolean expression } bexp \text{ is } false$$

This rule maintains that if the evaluation of the boolean expression is false then Q performs a transition α and jumps to the state Q'.

Let's illustrate these new rules by two examples inspired by the Peano arithmetic[2]. Consider the following definitions:

$$Succ \overset{def}{=} in(x)Succ'(x)$$

[2]https://en.wikipedia.org/wiki/Peano_axioms - 2017-03-20

$$Succ'(x) \stackrel{def}{=} \overline{out(x+1)}.Succ$$

Succ is a buffer that holds one number and computes its successor. Let n be a natural number received in input. When $Succ$ receives the number n, it binds its value to the parameter x. The process will then apply the second equation and behaves like $Succ'(n)$. The program output will be the evaluation of the expression $out(n+1)$. Finally the program will go back into its initial state.

Let us now consider to the following definitions :

$$Pred \stackrel{def}{=} in(x)Pred'(x)$$

$$Pred'(x) \stackrel{def}{=} if\, x = 0\, then\, \overline{out(0)}.Pred\ else\ \overline{out(x-1)}.Pred$$

Here, $Pred$ is a buffer that holds one number and computes its predecessor. Since we want to compute only natural numbers, a conditional clause has been added in order to avoid negative numbers.

As mentioned earlier, such language extension is unnecessary. We shall now demonstrate how our examples can be reduced to the basic calculus[3].

For the sake of simplicity, we consider all inputs to belong to a finite set V. If we take the parametrized constant $Succ'(x)$, we are able to transform it into a set of constants $Succ'_v$, one constant for each value of $v \in V$.

The same applies for the output prefix $\overline{out(x+1)}$, which now becomes a set of $\overline{out_{v+1}}$. Thus, our equation for $Succ'$ becomes a family of equations:

$$Succ'_v \stackrel{def}{=} \overline{out_{(v+1)}}.Succ\ (v \in V)$$

As for the prefix $in(x)$, in order to reflect that it can take any input from V and binds it to x. We can translate it to a sum $\sum_{v\in V} in_v$. The equation is now:

$$Succ \stackrel{def}{=} \sum_{v\in V} in_v.Succ'_v\ (v \in V)$$

As for the second example, $Pred$, the first equation has the following translation:

$$Pred \stackrel{def}{=} \sum_{v\in V} in_v.Pred'_v\ (v \in V)$$

[3]The interested reader can find the exhaustive conversion rules in [16, p. 55,56]

The second equation translates to a family of equations for each v but the right member of the equation depends on the nature of v:

$$Pred'_v \overset{def}{=} \begin{cases} \overline{out_{(v-1)}}.Succ \; if \; v \neq 0 \\ \overline{out_{(0)}}.Succ \; if \; v = 0 \end{cases} \quad (v \in V)$$

The internal boolean expression has thus become an external condition describing a family of equations.

2.5 Conclusion

In this chapter we presented an overview of CCS, a subject for which many books exist. We have selected one of them, Reactive Systems [1], that was helpfully didactic in guiding us in our journey to the exploration of the language. We have chosen to follow the same path in the present chapter and in Chapter 5 on Behavioral Equivalences.

3 Scala

The aim of this chapter is to introduce the reader to Scala. Our goal is not to be as exhaustive as a language specification could be, but rather to give the essential parts needed to understand the present work.

Most of the content of this chapter, is a condensed view derived from a technical report written by the authors of the language in 2006 [17].

Scala is a programming language designed and developed in the programming methods laboratory of the EPFL since 2001 [18]. Its creators noticed that in the software industry only a few components are actually reused. So Scala was built to address this limitation by basing it on two main principles: first that the same concept should be able to describe both small and large parts of a system and thus to grow as its applications demand (hence, the origin of its name). Second, that this scalability should be provided by a combination of functional and object-oriented paradigms.

3.1 A Java like language

Scala was built to easily interact with mainstream platforms such as C# or Java. Scala can natively use classes, libraries and frameworks from those platforms. Programs can reuse existing code previously developed in Java and conversely, Java seamlessly can use frameworks written in Scala.

For example, the Play framework was developed in Scala and can be used either in Java or in Scala [3]. Moreover, Scala shares most of its syntax with Java and C# which accelerates the learning curve to use it.

3.2 A Unified Object Model

"Scala uses a pure object-oriented model" [17, p. 3]. "Every value is an object and every operation is a message sent" [17, p. 3].

3.2.1 Classes

All classes belong to one of two groups: references or values. The reference classes are subtypes of *AnyRef* and the value classes are subtypes of *AnyVal*. *AnyVal* and *AnyRef* are subtypes of the super class *Any*. Note, that usually the reference classes are stored in the heap whereas the value classes are stored in the stack. Because all values are subtypes of *AnyVal*, there is no primitives in Scala.

3.2.2 Operations

As stated earlier, each Scala operation is a message sent via a method call. The operators are labeled as ordinary identifiers. If an operator is between two expressions, the compiler will consider it as a method call, allowing it to be considered as an infix operator. For instance, $a + b$ is equivalent to $a.+(b)$. This syntactic sugar can however be used only with one parameter.

Scala authorizes method calls without evaluating the parameters. This technique is called lazy loading. For achieving this, the compiler uses a call-by-name. Usually, Scala uses call-by-value but if the type of function parameter starts with =>, it uses a call-by-name. Let us consider the following snippet.

```
def f(value: Int, name: =>Int)= value

f(1, NonTerminatingComputation) //(a)

f(NonTerminatingComputation, 1) //(b)
```

In the definition of the function f, the left parameter is a call-by-value and will be evaluated when passed to the function. In contrast, the right parameter is call-by-name and the evaluation will be deferred until the point at which it is needed. In our case, the first call to f will return 1 while the second call to f will never terminate. This leads us to the following conclusion: "If call-by-value evaluation of an expression e terminates, then call-by-name evaluation of e terminates also. The other direction is not true." [19]

3.2.3 Variables and Properties

Variable dereferencing and assignment are method calls as well. They are performed through auto-generated getters and setters.

```
def x: T
def x_= (newval: T): unit
```

Actually, every reference to x is interpreted as a call to $def x$,and every assignment such as $x = 5$ is interpreted as $x.\_ = (5)$. The programmer has the possibility to redefine setters and getters to add properties on them.

```
class Natural{
   private x: Int
   def n: Int = x
   def n_=(y :Int):Unit = if(y<=0) x=0 else x=y
}
```

In the class Natural, every assignment verifies whether the value is positive. If not, the value is replaced by zero. Every reference simply returns the value of x.

3.3 Operations are Objects

3.3.1 Methods are Functional Values

Since methods are functional values, they can be assigned to a variable and passed as a function parameter. Functions are also able to return a function as a value. Those functions are called *Higher-order functions*. Scala offers the possibility to define a function without giving it a name. Those anonymous functions are called *lambdas*. To illustrate those concepts, let's have a look to the following code:

```
def find(as: List[A],p: A => Boolean):Option[A]= { //high-order function
   if (as.isEmpty) None
   else if (p(as.head)) Some(as.head)
   else find (as.tail,p)
}

val as = List(1,2,3,4)
//This high-order function will look up if 3 is present in the list and
    then return it.
```

```
find(as, x => x == 3) //lambda
```

find takes a list and predicate as parameters. In code here above, the predicate is illustrated as a *lambda*. The function *find* recursively iterates on the list. This function returns an optional value. *None* if nothing is found, and *Some* otherwise. The value found is returned wrapped by the class *Some*.

Higher-order functions offer us the possibility to use the same function for every possible predicate without rewriting code.

3.3.2 Functions are Objects

"If methods are values, and values are objects, it follows that methods themselves are objects"[17, p. 6] Function are actually a syntactic sugar for a specific type of class. For instance the function $S \Rightarrow T$ is actually the class:

```
package scala
abstract class Function[-S, +T]{
   def apply(x :S): T
}

val square : Int => Int = ( a :Int ) => a * a
square(2) //res1: Int = 4

val square2 : Int => Int = new Function[Int, Int]{
   def apply( a :Int ) = a * a
}
square2.apply(2) //res2: Int = 4
square2(2)       //res3: Int = 4
```

The function *square* is defined with a functional style without any reference to an object creation. But under the hood the compiler automatically transforms *square* into an instance of $Function[-S, +T]$ as illustrated by *square*2. Moreover, the name of the function followed by its arguments inside parenthesis is a syntactic sugar for a call to the function *apply*, e.g., *square*2 is interpreted as $square2.apply(2)$.

3.3.3 Refining Functions

Since functions are objects, they are able to use the inheritance mechanism.

```
class Array[T] extends Function1[int, T] with Seq[T] {
def apply(index: int): T = ..
...
}
val a: Array[Char] = List('a','b','c','d').toArray

a(3) //res1 = d idem as a.apply(3)
```

An example of function inheritance in the standard library are arrays. They inherit from $Function1[int, T]$. In the example above T is binded to $Char$ in the array declaration. As displayed in the last line of code, the array access is interpreted as a function application through the function *apply*. Because arrays inherit from $Function1$ they can use this syntactic sugar.

3.3.4 Sequences and For Comprehensions

There are several types of sequences in the main Scala library such as iterators, arrays, and streams. On those sequences, one can apply existing high-order functions such as filter or map. Scala offers the possibility to use for-loop in comprehension.

```
def processNegativeNumber(xs: List[Double]): List[Double] {
   xs filter (x => x < 0) map (x => -x)
}
def processNegativeNumber2(xs: List[Double]): List[Double]{
   for ( val x <- xs; 0 < x) yield -x
}
```

Both functions have exactly the same behavior. The first one uses high-order functions applied to a list. It filters the elements and then maps them to a new positive value.

The second one uses a *for* loop in comprehension. The first part contains the generator, the second part contains the filter. The keyword *yield* is used to generate a new value for each matching element. This new value is buffered until the end of the loop. Finally, the buffer is returned as a sequence.

3.4 Abstraction

Scala offers two types of abstraction, by parametrization (Functional) and by abstract member (Object Oriented).

3.4.1 Functional Abstraction

Classes can abstract over the type.

```
class ImmutableWrapper[T](init: T) {
   private val value: T = init
   def get: T = value
}

ImmutableWrapper[Int](5)
ImmutableWrapper("infered type")
```

For each usage, the class parameter T will be replaced by the actual type of the value given in the constructor. The *ImmutableWrapper* can be used with any type. This class is generic. Note that, the compiler has the ability to directly infer the type. Functions have the ability to be generic as well.

```
def clone[T](x: ImmutableWrapper[T]) : ImmutableWrapper[T] =
    ImmutableWrapper[T](x.get)

val x: ImmutableWrapper[Int] = ImmutableWrapper[Int](5)
val y = clone[ImmutableWrapper[Int]](x)
```

Generic type parameters can be bounded to a specific subtype. As the language supports the F-bounded polymorphism, the question of sub-typing variance is raised. Are they contra-variant, covariant, or non-variant ? The language permits all of those possibilities.

3.4.2 Abstract Members

As an alternative to abstraction by parametrization, object oriented abstraction can be used. This can be achieved via abstract type members. Of course an abstract type may have only one definition.

```
abstract class ImmutableWrapper[T](init: T) {
```

```
   type T
   val init: T
   private val value: T = init
   def get: T = value
}
val x = new ImmutableWrapper {type T = Int; val init = 5 }
```

3.4.3 Modeling generics with abstract types

The functional abstraction (generics) can be replaced by the oriented object abstraction (abstract members). It is therefore possible to achieve the same functionalities with only one paradigm, but the language will loose conciseness. Furthermore, "generics are typically used when one needs just type instantiation, whereas abstract types are typically used when one needs to refer to the abstract type from client code." [17, p. 11].

3.5 Composition

The mechanism used in Scala to compose classes are *mixins*. Mixins are a solution to multiple inheritance. Therefore, with this solution a class can inherit from several traits. A trait is an interface that may implement methods. Only traits can be used in a mixin. Let us illustrate this with a foretaste of the next chapter.

```
trait CCS

trait Composition {
  this: Process =>

  def |(other: Process) = CompositionProcess(this, other)
}

trait Summation {
  this:Process =>

  def +(other: Process) = SummationProcess(this, other)
}

class Process extends CCS with Summation with Composition
```

```
val x :Process = new Process
val y :Process = new Process

val summationP = x + y
val compositionP = x | y
```

As a result of the mixin composition, *Process* has two operations available + and |.

If two traits have methods with the same signatures, the compiler takes the delta between the methods of the super class and the ones from the traits based on the order of apparition after the keyword *with*.

Inherited methods can be overloaded if they have the same signature. As in Java, the lowest definition is always picked. If no implementation exists for a method, the class or trait is abstract otherwise it is said as concrete.

3.6 Decomposition

In object oriented programming, structured data can only be decomposed through a set of methods directly implemented in the data structure. To avoid class modification and separate the algorithm from the structure, one can use the visitor pattern [1]. But Scala proposes a functional way of achieving decomposition with *pattern matching*.

```
def sum(l: List[Int]): Int = l match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

The function *sum* recursively iterates over the list l and returns the sum of the elements. The block with the case statements tries to find a match for the list l received in parameter. It uses the policy of first case matching. Once a matching is found the statement of the case is applied. The first case is the base case. It matches an empty list and returns zero. The second case is the inductive case which sums every element of the list.

[1] The interested reader can found more details about the visitor pattern: `https://en.wikipedia.org/wiki/Visitor_pattern`

3.7 XML Processing

Scala has been designed to ease maintenance and construction of programs that use XML data format. The XML syntax can be natively used in the language. The decomposition of the XML structure is done through pattern matching. Searches can be easily done with for-loop in comprehension with a style close to XQuery.

3.8 Component Adaptation

Integration of existing components can be difficult. Let us suppose one wants to integrate an existing library in one's code base. The interface offered by this library may not meet one's requirements. An object oriented approach to this problem is either to use inheritance or a design pattern such as the visitor. Those solutions can be complex to implement and to maintain [24]. "This unsatisfactory situation is commonly called the external extensibility problem" [17, p. 15].

Scala has introduced *views* to solve the external extensibility problem. *Views* are a special case of *implicit parameters*.

3.8.1 Implicit Parameters

To illustrate the concept of implicit, let us start with a concrete example. Imagine a class with a set of methods doing asynchronous calls to a web-service. In order to do so, one has to pass around an *ExecutionContext* in each of these methods. Of course, this can be achieved manually but in a large code base with a lot of parameters, this can result in a lot of boilerplate code.

To avoid passing the same parameter over and over in each method calls, Scala proposes to use implicit parameters. When a method has a parameter declared as implicit, the compiler tries to infer the missing implicit parameter from the scope. If there are several choices, the compiler chooses the correct one with the same rules as the overloading.

```
implicit val ec: ExecutionContext =
    scala.concurrent.ExecutionContext.Implicits.global

def getSoldier(id: Int)(implicit e: ExecutionContext): Future[Soldier] =
    ???
def getRank(s :Soldier)(implicit e: ExecutionContext): Future[Rank] = ???
```

```
val rankedSoldier: Future[SoldierWithRank] =
  getSoldier(100).flatMap { s =>
    getRank(s).map { r =>
      SoldierWithRank(s.id, s.name,r)
    }
  }
}
```

Let us imagine the functions *getSoldier* and *getRank* have to fetch their data via a web-service. For performance reasons, they performs asynchronous calls. Because *ExecutionContext* is declared as an implicit, it is automatically provided to *getSoldier* and *getRank*. This is illustrated in the last statement: the two functions are called to retrieve the soldier with the id 100, along with his rank, without *ExecutionContext*.

3.8.2 Views

"Views are implicit conversions between types" [17, p. 17]. Views can be used to add extra functionalities on existing classes without altering them.

```
implicit class ImprovedInt(i :Int){
   def incr = i + 1
}

val counter = 5
val next = counter.incr
```

ImprovedInt adds extra behavior on *Int*. This class is defined as an implicit. First, the compiler treats the variable *counter* as a regular *Int*. But when the method *incr* is called on *counter*, the compiler converts to the implicit type *ImprovedInt* because the method *incr* is not available on *Int*.

3.9 Conclusion

Scala has a rich syntax, a convenient inference type system and combines object-oriented and functional paradigms. However, Scala can be perceived as simple, as it is based on just a few principles. Its language constructs allow us to abstract

and create new components. Scala promises to provide the necessary tools for building reusable pieces of code.

4 Implementation

4.1 Workbench requirements

In the introduction we provided a general description of our tool. It is now the time to define it.

The first goal of our tool is to model CCS equations as graphs. Those graphs allow us to visualize intercommunication and potential deadlocks. We also want to interact with graphs and explore step by step equations.

It follows that the workbench has to provide a way to input CCS formulae and display them as graph in two modes:

- a discovery view, where the user clicks on the graph nodes to expand the equation step by step;
- a static view, where the equation is already expanded.

A desirable property is that the graph should be an unmodified transcription of the equation. It means that one should be able to deduce the original equation from the graph.

The second goal of our tool is to compare CCS equations through a range of behavioral equivalence and preorder properties.

To do so, the tool has to provide a way to select two CCS equations and apply the desired property check on them. Ideally, the tool will provide us meaningful details on the check.

So far, we have introduced CCS and Scala. That gives us enough material to start

implementing the first goal of the tool.

As for the second goal, the checks will be directly applied on the graph previously generated by the first step. But before explaining their algorithms and implementations, theoretical notions on equivalences should be introduced. For that reason, we have chosen to detail all those aspects in a separate chapter (Chapter 5 Behavioral Equivalences).

4.2 Implementation overview

Let us give an overview of the steps needed to implement the first part of our tool.

- The first step is the parsing. The goal of this step is to provide a convenient data structure on which we can perform computations. The produced data structure is an abstract syntactic tree - AST;
- The second step is to generate a set of transitions for an equation;
- Then from those transitions we will generate a graph and display it in the view.

The scheme illustrated in Figure 4.1 will guide our development and will be detailed in the next sections.

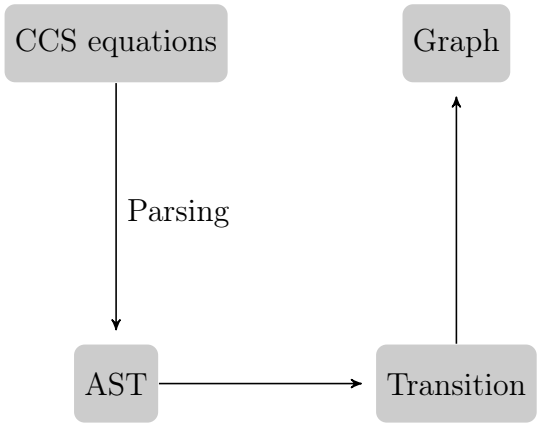


Figure 4.1: Implementation diagram

4.3 Parsing

In this section, we will process the formula received in input and analyze it. This analysis will allow us to create an abstract syntactic tree. To transform the flat text received into an AST, few solutions exist. One of them is the combinator parsing.

4.3.1 Combinator Parsing

To build this type of parser, we use primitive parsers and combinators. The role of the primitive parser is to recognize the subsequences of our text. The role of the combinators is to combine the parsers between them to build a more complex one. The result of this combination is a new parser that itself can be combined with others.

The combinators are just functions that can apply [12, p. 13]:

- Repetitions: apply multiple times the parser to the input (zero or more, on or more, zero or one);
- Sequence: each parser has to successively succeed according its position in the sequence;
- Alternation: at least one of the parsers must validate the sub-sequence.

In this approach, the lexical and syntactical analysis are unified in the very same grammar. But why would one choose this approach instead of the traditional method where the analysis is performed in two distinct phases ? There are two reasons described in the following subsections.

Scannerless

In the classical approach, the lexical analyzer, lexer, uses regular expression to create tokens. Those tokens are then given to the syntactic analyzer. This analyzer applies a context free grammar (CFG) to each token to build the data structure. However, the lexer may need a context to decide which component to create. For instance, in Java the character $<$ could be either a comparison operation as in $a < b$, or can be used to write a generic type such as in $List < T >$.

The lexer may also need to recognize nested structures. To decide which rule to apply depending on the context, some priority rules have to be defined. This mechanism can be complicated. On the other hand, in the scannerless approach,

one has access to the context at any moment. Therefore, the priority rules are no longer relevant. This is the standard provided method by Scala combinator Parser [12, p. 13, 19, 20].

Parsing Expression Grammar

Another difference between the two approaches is the grammar. The implementation of our parser uses a parsing expression grammar - PEG instead of a context free grammar - CFG.

The context free grammars were developed to recognize the natural languages. However, those grammars may not be relevant for parsing formal languages. They are generative, which means that they define rules on how to produce a word of the language but not how to identify it.

A second drawback of CFG is that they are prone to ambiguity. The same input can have several possible derivations, which makes sense in a natural language but not for a formal language. To resolve that, some rules have to be added to the grammar which complicates it.

On the other hand, a PEG is based on recognition instead of generation. This resolves ambiguities via deterministic priority rules. This grammar combines the EBNF annotation and regular expression, to allow a more natural transposition of the source language grammar. The PEG grammar is used by the standard combinator parser provided by Scala [12, p. 13, 19, 20].

4.3.2 Abstract Syntax Tree - AST

The data structure created by the parser is an abstract syntactic tree. Each node of the tree denotes a construct occurring in CCS. The internal nodes depict the different operators available in CCS. The leaves are either an action or the process Nil.

An implementation of $\pi - Calculus$ [1] as a domain specific language[2] has been proposed by Matiello Pedro[14]. We have used that structure as a guideline and adapted it to our domain.

[1] Pi-Calculus is the continuation of CCS: `https://en.wikipedia.org/wiki/%CE%A0-calculus`

[2] A language applied to a specific domain. The interested reader can find more information at `http://www.scala-lang.org/old/node/1403`

Figure 4.2 shows the class hierarchy used to build the tree. It is followed by a detail of the implementation of each classes. For the sake of clarity, only the relevant methods are displayed. Finally, few examples to illustrate the interlacing of nodes and leaves.

```
trait CCS
```

This trait is at the summit of the hierarchy. It allows us to type all the objects belonging to CCS.

```
trait Process extends CCS with Summation with Composition with
    Restriction
```

Process is a CCS expression on which one can apply choices(+), composition(|), or restriction (\) operators. As the astute reader can remember from the chapter on Scala, the mixin technique is used to provide extra functionalities.

Operations

```
trait Summation {
  this:Process =>

  def +(other: Process) = SummationProcess(this, other)
}

trait Restriction {
  this:Process =>

  def \(other: => List[InputAction]) = RestrictionProcess( this, other)
}

trait Composition {
  this: Process =>

  def |(other: Process) = CompositionProcess(this, other)
}
```

- Summation provides us a method + to sum two processes. As a result, it creates a node of type *SummationProcess* composed from the two expressions.

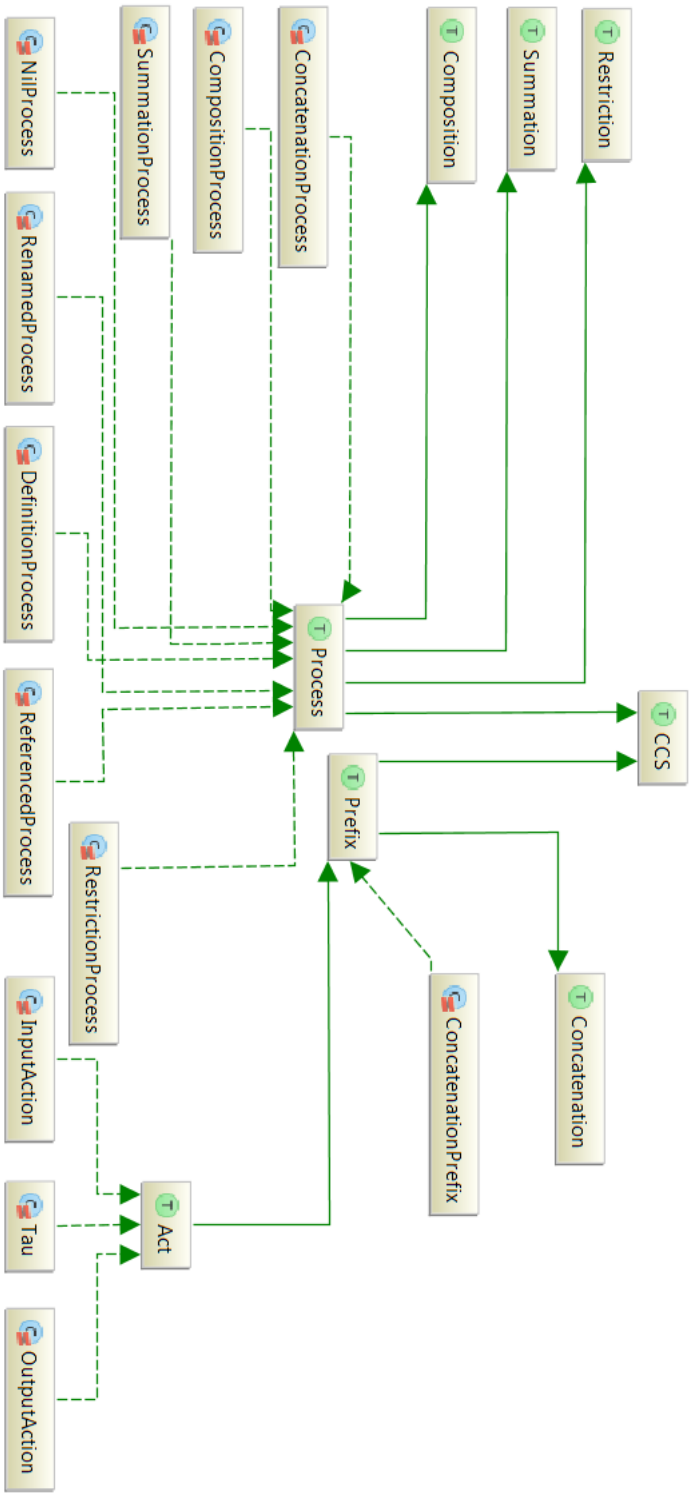


Figure 4.2: AST Diagram

- Restriction provides us a method \ to restrict a process. As a result, it creates a node of type *RestrictionProcess* composed with a process and a list of restricted actions.
- Composition provides us a method | to compose two processes. As a result, it creates a node of type *CompositionProcess* composed from the two expressions.

Those newly created objects permit us to write code that is close to the syntax of CCS:

```
val C = A | B    // A and B are defined as process
val D = A + B
val E = C \ xs   // xs is defined as a list of actions
```

A last point to highlight is the usage of the self-type reference i.e: $this : Process =>$ With this technique, each object that instantiate *Summation* has to be mixed with the trait *Process*.

Hence, *Summation* has access to *Process*, which allows the method + to create a *SummationProcess* with a reference to the process given as a parameter and itself (this).

This style handles the dependency between two traits without resorting to inheritance. This is an elegant way of doing component injection without a framework such as spring.

Let us turn to the classes implementing those traits.

Processes

```
case class DefinitionProcess(id:String, definition:Process) extends
    Process

case class ConcatenationProcess(val left: Prefix, val right: Process)
    extends Process

case class SummationProcess(left: Process, right: Process) extends
    Process

case class CompositionProcess(left: Process, right: Process) extends
    Process
```

```
case class NilProcess() extends Process

case class RestrictionProcess(process: Process, other:
    List[InputAction]) extends Process

case class ReferencedProcess(id: String) extends Process

case class RenamingProcess(process: Process, other: List[(InputAction,
    InputAction)]) extends Process
```

Those classes are defining all the different processes available in CCS. There is a slight difference between $DefinitionProcess$ and $ReferencedProcess$.

$DefinitionProcess$ symbolizes a CCS equation. The id and $definition$ parameters are respectively the left and the right side of the CCS equation. $ReferencedProcess$ is a pointer to an equation that may not be defined yet.

This allows us to deal with forward process definition such as :

$$A = a.B$$

$$B = b.d.0$$

Action prefixing

```
trait Prefix extends CCS with Concatenation

trait Concatenation {
  this: Prefix =>

  def *(other: => Process) = ConcatenationProcess(this, other)
}

case class ConcatenationPrefix(actions: List[Prefix]) extends Prefix
```

- $Prefix$ allows us to identify all expressions on which we are authorized to apply action prefixing.
- $Concatenation$ provides us the action prefixing through its method $*$.
- $ConcatenationPrefix$ is a sequence of prefixed actions. Instead of list of $Prefix$ we could have used a nested structure of $Prefix$. But the list struc-

ture allows us to know the previous and next action easily. This will simplify the transition generation in the next steps.

Actions

```
trait Act extends Prefix
case class InputAction(id:String) extends Act
case class OutputAction(id:String) extends Act
case class Tau(id:String) extends Act
```

Those classes depict the three kind of actions and they are all subtype of $Prefix$. Let's turn now to the following examples,

$$\begin{cases} 1 & = & P \stackrel{def}{=} \overline{push}.ticket.\overline{money}.P \\ 2 & = & TM \stackrel{def}{=} push.\overline{ticket}.TM \\ 3 & = & PH \stackrel{def}{=} (TM|P)\backslash\{push,\ ticket\} \\ 4 & = & VM \stackrel{def}{=} coin.(\overline{chocolate}.\emptyset + \overline{candy}.\emptyset) \end{cases}$$

To provide those equations as a flat text to the parser, one has to input them in a syntax supported by the workbench. To do so, they have to be rewritten according to the syntax defined in Table 6.1. The four equations become :

$$\begin{cases} 1 & = & P = \_push.ticket.\_money.P \\ 2 & = & TM = push.\_ticket.TM \\ 3 & = & PH = (TM|P)\backslash\{push, ticket\} \\ 4 & = & VM = coin.(\_chocolate.0 + \_candy.0) \end{cases}$$

Afterwards, the parser produces the following results :

```
//1
val P = ProcessDefinition("P",
    ConcatenationProcess(
        ConcatenationPrefix(
            List(OutputAction("push"),InputAction("ticket"),OutputAction("money")),
    ReferencedProcess("P"))
//2
val TM = ProcessDefinition("TM",ConcatenationProcess(
ConcatenationPrefix(List(InputAction("push"),OutputAction("ticket"))),
    ReferencedProcess("TM"))
//3
```

```
val PH = ProcessDefinition(PH, RestrictedProcess(
CompositionProcess(P.definition,TM.definition),
    List(InputAction("push"),InputAction("ticket"))
//4
val VM = ProcessDefinition("VM", ConcatenationProcess(
ConcatenationPrefix(List(InputAction("coin")),
    SummationProcess(
      ConcatenationProcess(ConcatenationPrefix(List(OutputAction("chocolate")),
      NilProcess),
      ConcatenationProcess(ConcatenationPrefix(List(OutputAction("candy")),
      NilProcess))))
```

4.3.3 Parsing Implementation

```
trait CCSParser extends JavaTokenParsers
```

CCSParser contains all the required parsing functions. It inherits from *JavaTokenParsers*, one of the combinator parser provided by the Scala standard library. Let us describe its main functions:

- ~: defines a sequence of parsers;
- .? or *opt* : defines an optional parser;
- | is: tries to match the left expression. If not it tries to match the right expression;
- $rep(p :\Rightarrow Parser[T])$: takes a parser as an argument and repetitively applies it until it fails. The result is a parser containing a list.
- $repsep[T](p :\Rightarrow Parser[T], q :\Rightarrow Parser[Any])$: takes two interleaving parser, P and Q, and tries to match them repetitively. The result is a parser containing a list of P elements.
- ^^ : applies a converting function to the result of the parser located on its left;
- ^^^ : directly replace the result of the parser located on its left by the element located on the right.

Let us now implement the parser. To do so, two fashions are available: top-down decomposition or bottom-up composition. Note that as printed out in [12, p. 28],

"Writing out grammar rules in a top-down or a bottom-up order is merely a matter of taste".

We will write our grammar rules in a top-down fashion. This means that we will start with the entire input on the top rule. Then from the top rule we will apply successive definitions which are going to lead us to the terminal rules. The succession of grammar rules will reflect the precedence of CCS operations. The loosest precedence is on top, the tightest is at the bottom :

- The first rule starts with the whole input and breaks it into an identifier followed by an Expression;

```
def parseProcess: Parser[Process] = processIdentifierRule ~ "=" ~
    Expression ^^ {
  case (x ~ "=" ~ y) => DefinitionProcess(x,y)
}
```

- An *expression* is a *summationRule*.

```
def expression: Parser[Process] = summationRule
```

- A *summationRule* is a *compositionRule* potentially followed by multiple *compositionRule* separated by "+". *summationRule* binds more loosely than other operators;

```
def summationRule: Parser[Process] = compositionRule ~ rep("+" ~
    compositionRule) ^^ {
  case (x ~ y) => y.foldLeft(x) {
    case (x, "+" ~ y) => x + y
  }
}
```

- A *compositionRule* is a *composedProcessRule* potentially followed by multiple *composedProcessRule* separated by "|". *compositionRule* binds more tightly than "+" but more loosely than other operators;

```
def compositionRule: Parser[Process] = composedProcessRule ~
    rep("|" ~ composedProcessRule) ^^ {
  case (x ~ y) => y.foldLeft(x) {
    case (x, "|" ~ y) => x | y
  }
}
```

- A *composedProcessRule*: is a *processRule* potentially followed by a *restrictionRule*

or a *renamingRule*. It can as well match an *expression* between parenthesis potentially followed by a *restrictionRule*;

```
def composedProcessRule: Parser[Process] = processRule ~
    opt(restrictionRule | renamingRule) ^^ {
    case x ~ y => restrictionOrRenamingProcessFactory(x, y)
  } | "(" ~ Expression ~ ")" ~ opt(restrictionRule) ^^ {
    case "(" ~ x ~ ")" ~ y => restrictionOrRenamingProcessFactory(x,
        y)
  }
```

- A *processRule* matches a set of actions separated by "." and followed by a *composedProcessRule*. This definition can also match *simpleProcessRule*;

```
def processRule: Parser[Process] = repsep(actionRule, ".") ~ "."
    ~ composedProcessRule ^^ {
  case x ~ "." ~ y => ConcatenationPrefix(x) * y
} | simpleProcessRule
```

- A *renamingRule* is a set of tuples. This tuple is composed by two input actions separated by "/";

```
def renamingRule: Parser[List[(InputAction, InputAction)]] = "["
    ~ repsep(inputActionRule ~ "/" ~ inputActionRule, ",") ~ "]"
    ^^ {
  case "[" ~ x ~ "]" => x.map({ case x ~ "/" ~ y => (x, y) })
}
```

- A *restrictionProcessRule* is the character "\" followed by a set of input actions;

```
def restrictionRule: Parser[List[InputAction]] =
"""\{""" ~ repsep(inputActionRule, ",") ~ "}" ^^ {
  case _ ~ x ~ _ => x;
}
```

- A *actionRule* is an input action or output action;

```
def actionRule: Parser[Prefix] = inputActionRule | outputActionRule
```

- A *simpleProcessRule* is an empty process or process identifier;

```
def simpleProcessRule: Parser[Process] = emptyProcessRule |
    processIdentifierRule
```

- A *processIdentifierRule* is terminal element defined as a set capital letters;

```
def processIdentifierRule: Parser[Process] = """[A-Z]+""".r ^^ {
    x => ReferencedProcess(x)
  }
```

- A *emptyProcessRule* is terminal element defined as the number "0";

```
def emptyProcessRule: Parser[Process] = """0""".r ^^^ NilProcess()
```

- An *inputAction* is terminal element defined as a set of lower-case letters;

```
def inputActionRule = """[a-z]+""".r ^^ { x => InputAction(x) }
```

- An *outputAction* is terminal element defined as a set of lower-case letters that stars with the character "_".

```
def outputActionRule ="""\_[a-z]+""".r ^^ { x =>
    OutputAction(x.substring(1)) }
```

One can notice that *summationRule* will only produce a *SummationProcess* if the right side of the "+" matches at least one time. Otherwise, the creation of the AST node will be defered to *composedProcessRule*, the function one level below. Each definition will proceed in the same fashion until they reach a terminal definition.

The methods ^^^ and ^^ are used to build the AST nodes. Detailing all the rules used in the parser will be tedious, but nevertheless we think describing the whole behavior of *summationRule* is of interest. Functional decomposition, compiler inference, high-order function and infix notation provided by Scala are illustrated in that snippet.

In *summationRule*, if the rule complies with the definition, pattern matching is applied to break down the equation. x is the first element and y the list of the remaining elements. We used the $foldLeft$ function to handle formula with multiple + as in: "A+B+...+N". In this case x is "A" and y is "B+...+N".
Thanks to the function return type, Scala compilers will infer the type of x and treats it as a *Process*.
$foldLeft$ takes an accumulator x and a function applied from left to right to each list element. This function will sum up two by two the processes with the function +. The function "+" is defined on the trait *Summation* from our AST. The result will be:

$$SummationProcess(A, SummationProcess(B, SummationProcess(...Process(N))))$$

```
case (x ~ y) => y.foldLeft(x) {
   case (x, "+" ~ y) => x + y
 }
```

4.4 Transitions

Now that we have our AST, we can compute the transitions for each process. A transition is defined by the following class :

```
case class Transition(source:Process, action: Prefix, target: Process)
```

An instance of a transition depicts the links between two states of an LTS. Let's remember the example from the section 2.2 LTS.

$$P \overset{def}{=} \overline{push}.ticket.\overline{money}.P$$

which gives the following transitions between states,

$$P \xrightarrow{\overline{push}} P1 \xrightarrow{ticket} P2 \xrightarrow{\overline{money}} P$$

Accordingly the following subsequent transitions will be generated from the AST:

```
Transition(P, OutputAction(push), P1)
Transition(P1, InputAction(ticket), P2)
Transition(P2, OutputAction(money), P)
```

Now that we have transitions and an AST, the next step will be to explain how to traverse the tree and build the transitions for each node encountered.

But first let's speak about case classes and pattern matching. We have already met case classes previously in the definition of the syntactic tree. According to the Scala documentation [5] case classes are:

- Immutable by default
- Compared by structural equality instead of by reference
- Succinct to instantiate and operate on
- Decomposable through pattern matching

In order to achieve the last point the compiler automatically generates an extractor method for the case classes. The purpose of the extractor is to pull out the inputs that were given in the constructor. This will allow the pattern matching to decompose the object.

Let's have a look to the function that travels across the tree :

```
def dispatcher(currentProcess: Process,
    restrictedActions:Option[List[InputAction]] = None):
    List[Transition] = {
  currentProcess match {
    case DefinitionProcess(left, right) => dispatcher(right)
    case c: ConcatenationProcess =>
       generateConcatenationTransition(c,restrictedActions)
    case s: SummationProcess => generateSummationTransition(s);
    case c: CompositionProcess => generateCompositionTransition(c,
       restrictedActions)
    case RestrictionProcess(left, right) => dispatcher(left,
       Some(right));
    case _ => List()
  }
}
```

This function goes through the AST and generates the adequate *Transitions* for each internal step of the equation. The result is a list containing all of the transitions.

As one can notice from Figure 4.2 which illustrates the AST, not all of the cases of processes are covered. Actually for the sake of simplicity, *RenamedProcess* is present in the parser but not supported further in our code. $ReferencedProcess$ is not there either. As we said earlier this function only builds the internal transitions for each process. The reason of this limitation is to avoid infinite computations in case of recursive process e.g.:

$$\begin{cases} A = a.b.B \\ B = d.e.A \end{cases}$$

The result for the $ReferencedProcess$, will be therefore matched by _ and an empty list will be returned. This is also true for $NilProcess$ and *RenamedProcess*.

The link between each process transition will be done in the graph generation. This will be our next step. However before going further, let us explain the transition generation for each managed process.

Concatenation Transition

```
def generateConcatenationTransition(currentProcess:
    ConcatenationProcess, restrict: Option[List[InputAction]] = None):
    List[Transition] = {
  val (left, right) = (currentProcess.left, currentProcess.right)
  // If only one action remains on the prefix, you have to link this
      action to the next Process when
  // creating the transition. example : a.B = Transition(a.B, a, B)
  if (left.size == 1) {
    right match {
      case NilProcess() => List(Transition(currentProcess,
          left.nextAction, right))
      case r: ReferencedProcess => List(Transition(currentProcess,
          left.nextAction, r))
      case c: ConcatenationProcess => Transition(currentProcess,
          left.nextAction, right) :: dispatcher(c, restrict)
      case s: SummationProcess => Transition(ConcatenationProcess(left,
          right), left.nextAction, right) :: dispatcher(s, restrict)
      case c: CompositionProcess => Transition(ConcatenationProcess(left,
          right), left.nextAction, right) :: dispatcher(c, restrict)
      case r: RestrictionProcess => Transition(ConcatenationProcess(left,
          r.process), left.nextAction, r.process) :: dispatcher(r)
      case _ => List()
    }
  }
  else {
    Transition(currentProcess, left.nextAction,
        ConcatenationProcess(left.nextStep, right)) ::
        generateConcatenationTransition(ConcatenationProcess(left.nextStep,
        right))
  }
}
```

The *else* branch of this function creates a transition between two actions. The new transition is added to a list. The tail of the list is generated by a recursive call to *generateConcatenationTransition*. These recursive calls travel the list of actions until only one remains.

When only one action remains the *if* branch creates a new transition from the last action to the next process. Depending on the type of the target process this transition varies.

Finally, in all cases but $NilProcess$ and $ReferencedProcess$, the dispatcher is called to generate the ad-hoc transitions for the next process. The resulting list of transitions is concatenated with the newly created transition.

Except the $RenamedProcess$, all the process type are covered. This function implements rule 2.8 ACT from the CCS chapter.

Summation Transition

```
def generateSummationTransition(summationP:
    SummationProcess):List[Transition] = {
   //creates left side transitions
   dispatcher(summationP.left).map(
      y=> Transition(SummationProcess(y.source,summationP.right),y.action,
      SummationProcess(y.target,summationP.right)))
   ++
   //creates right side transitions
   dispatcher(summationP.right).map(
      y=>Transition(SummationProcess(summationP.left,y.source),
      y.action,SummationProcess(summationP.left,y.target)))
}
```

This function implements the rule 2.10 SUM. First, the *dispatcher* is called on the left side of the + and returns a list of transitions. Then, we consume the list received by moving forward each transition and link them to the right side. Let us illustrate this by the next process:

$$a.b.\emptyset + l.k.\emptyset$$

The left side of the equation gives us the subsequent transition set :

$$dispatcher(a.b.\emptyset) \begin{cases} (a.b.\emptyset,\ a,\ b.\emptyset) \\ (b.\emptyset,\ b,\ \emptyset) \end{cases}$$

Then, the map function is applied:

$$\begin{cases} ((a.b.\emptyset + l.k.\emptyset),\ a,\ (b.\emptyset + l.k.\emptyset)) \\ ((b.\emptyset + l.k.\emptyset),\ b,\ (\emptyset + l.k.\emptyset)) \end{cases}$$

Those transitions include all the valid steps for the left side. The second part of the algorithm builds the mirror steps for the right side. Finally, the two lists are merged.

Composition Transition

```
def generateCompositionTransition(c: CompositionProcess)(implicit
    restrict: Option[List[InputAction]]): List[Transition] = {

   def _COM1(currentProcess: CompositionProcess): List[Transition] =
      {...}

   def _COM2(currentProcess: CompositionProcess): List[Transition] =
      {...}

   def _COM3(currentProcess: CompositionProcess): List[Transition] =
      {...}

   _COM1(c) ++ _COM2(c) ++ _COM3(c)
 }
```

generateCompositionTransition implements the transition rules $COM1$, $COM2$, $COM3$ defined in 2.11, 2.12, 2.13. It also handles the restriction RES defined by the rule 2.15. Each call to $\_COM1$, $\_COM2$, $\_COM3$ produces a list of transitions, then the lists are concatenated.

We will go through each of them and illustrate them with the following example:

$$a.b.\emptyset \mid a.\bar{b}.\emptyset$$

The careful reader can remember that this example was illustrated in Figure 2.6. As one may notice, *generateCompositionTransition* should produce a set of 13 transitions.

Let's start with $\_COM1$.

COM1

```
def _COM1(currentProcess: CompositionProcess): List[Transition] = {
     val (left, right) = (currentProcess.left, currentProcess.right)
     if (left.size() != 0 &
         !(left.getNextAction().isInstanceOf[NilProcess])) {
       left match {
         case ConcatenationProcess(x, y) =>
           if (isNotRestricted(x.nextAction, restrict)) {
```

```
            Transition(currentProcess, x.nextAction,
                CompositionProcess(ConcatenationProcess(x.nextStep, y),
                right)) ::
              generateCompositionTransition(
              CompositionProcess(ConcatenationProcess(x.nextStep, y),
                 right))
          }
          else List()
        }
      }
      else List()
    }
```

$\_COM1$ tries to create a transition by doing a step to the left, if the action is not restricted. For our example the first transition produced is :

$$a.b.\emptyset \mid a.\overline{b}.\emptyset \xrightarrow{a} b.\emptyset \mid a.\overline{b}.\emptyset$$

If the transition is built, it recursively calls $generateCompositionTransition$ with a sub equation that does not contain the left action e.g.: $b.\emptyset \mid \overline{a}.d.\emptyset$. Finally, $generateCompositionTransition$ calls $\_COM1$, $\_COM2$, $\_COM3$ on that sub-equation.

$\_COM1$ produces the 6 following equations :

$$\_COM1 \begin{cases} a.b.0 \mid a.\overline{b}.0 \xrightarrow{a} b.0 \mid a.\overline{b}.0 \\ b.0 \mid a.\overline{b}.0 \xrightarrow{b} 0 \mid a.\overline{b}.0 \\ a.b.0 \mid \overline{b}.0 \xrightarrow{a} b.0 \mid \overline{b}.0 \\ a.b.0 \mid 0 \xrightarrow{a} b.0 \mid 0 \\ b.0 \mid 0 \xrightarrow{b} 0 \mid 0 \\ b.0 \mid \overline{b}.0 \xrightarrow{b} 0 \mid \overline{b}.0 \end{cases}$$

COM2

```
def _COM2(currentProcess: CompositionProcess): List[Transition] = {
    val (left, right) = (currentProcess.left, currentProcess.right)
    if (right.size() != 0 &
        !(right.getNextAction().isInstanceOf[NilProcess])) {
      right match {
```

```
        case ConcatenationProcess(x, y) =>
          if (isNotRestricted(x.nextAction, restrict)) {
            Transition(currentProcess, x.nextAction,
                CompositionProcess(left,
                ConcatenationProcess(x.nextStep, y))) ::
              generateCompositionTransition(CompositionProcess(left,
                 ConcatenationProcess(x.nextStep(), y)))
          } else List()
      }
    }
    else List()
  }
```

$\_COM2$ tries to create a transition by doing a step to the right, if the action is not restricted. The first transition produced by $\_COM2$ is :

$$a.b.\emptyset \mid a.\overline{b}.\emptyset \xrightarrow{a} a.b.\emptyset \mid \overline{b}.\emptyset$$

If the transition is built, it will recursively call $generateCompositionTransition$ with a sub equation that does not contain the right action e.g.: $a.b.\emptyset \mid d.\emptyset$. Finally, $generateCompositionTransition$ calls $\_COM1$, $\_COM2$, $\_COM3$ on that sub-equation.

$\_COM2$ produces the 6 following equations :

$$\_COM2 \begin{cases} a.b.0 \mid a.\overline{b}.0 \xrightarrow{a} a.b.0 \mid \overline{b}.0 \\ b.0 \mid a.\overline{b}.0 \xrightarrow{a} b.0 \mid \overline{b}.0 \\ 0 \mid a.\overline{b}.0 \xrightarrow{a} 0 \mid \overline{b}.0 \\ a.b.0 \mid \overline{b}.0 \xrightarrow{\overline{b}} a.b.0 \mid 0 \\ 0 \mid \overline{b}.0 \xrightarrow{\overline{b}} 0 \mid 0 \\ b.0 \mid \overline{b}.0 \xrightarrow{\overline{b}} b.0 \mid 0 \end{cases}$$

COM3

```
def _COM3(currentProcess: CompositionProcess): List[Transition] = {
    val (left, right) = (currentProcess.left, currentProcess.right)
    if (left.size() != 0 &
        !(left.getNextAction().isInstanceOf[NilProcess]) &&
        right.size() != 0 &
        !(right.getNextAction().isInstanceOf[NilProcess])) {
```

```
      (left, right) match {
        case (ConcatenationProcess(ll, lr), ConcatenationProcess(rl,
            rr)) =>
          val (leftAction, rightAction) = (ll.nextAction, rl.nextAction)
          if (leftAction.complementaire(rightAction)) {
            Transition(currentProcess, Tau(),
                CompositionProcess(ConcatenationProcess(ll.nextStep(),
                lr), ConcatenationProcess(rl.nextStep(), rr))) ::
              generateCompositionTransition(
              CompositionProcess(ConcatenationProcess(ll.nextStep(),
                  lr), ConcatenationProcess(rl.nextStep(), rr)))
          }
          else List()
        }
      }
      else List()
    }
```

$\_COM3$ tries to create a transition for complementary actions, if they are not restricted. Actions are complementary iff they are input and output actions with the same label. The first transition produced by $\_COM3$ is :

$$b.0 \mid \overline{b}.0 \xrightarrow{tau} 0 \mid 0$$

If the transition is built, it will recursively call $generateCompositionTransition$ with a sub equation that does not contain the right action e.g.: $b.\emptyset \mid d.\emptyset$. Finally, $generateCompositionTransition$ calls $\_COM1$, $\_COM2$, $\_COM3$ on that sub-equation. The transition produced earlier is also the only one created by $\_COM3$:

$$\_COM3 \left\{ b.0 \mid \overline{b}.0 \xrightarrow{tau} 0 \mid 0 \right.$$

Restriction

```
def dispatcher(currentProcess: Process,
restrictedActions:Option[List[InputAction]] = None): List[Transition]={
...
case RestrictionProcess(left, right) => dispatcher(left, Some(right))
}
```

This case initialized the optional $restrictedActions$ with a list of actions.

4.5 Graph generation

As said in section 4.1, one of the main goal of our tool is to model CCS equations as graphs. To do so, we will use the graph framework *vis.js*.

4.5.1 vis.js

vis.js is an open source JavaScript library: "A dynamic, browser based visualization library. The library is designed to be easy to use, to handle large amounts of dynamic data, and to enable manipulation of and interaction with the data"[3].

We decided to use ***vis.js***, because it's well documented and it has an active online community. Furthermore, what tipped the scale in its favor was our ability to quickly build a prototype that fits our needs. We will review the advantages and disadvantages of the *vis.js* framework in the conclusion.

Creating a graph with vis requires specifying nodes and edges. This can be achieved with two DataSets. Furthermore, the DataSet structure provided by *vis.js* allows dynamic data binding.

Once the DataSets are initialized, all one needs is a HTML div container to put the graph in. Additionally one can use an options object to customize the graph such as auto-resize, height, width... Let's look at the code:

```
// create an array with nodes
 var nodes = new vis.DataSet([
   {id: 0, label: 'A'},
     {id: 1, label: '1'},
     {id: 2, label: '2'},
     {id: 3, label: '3'}
 ]);

 // create an array with edges
 //'arrows' enable directed graph
 var edges = new vis.DataSet([
     {from: 0, to: 1, label: 'a', arrows:'to'},
     {from: 1, to: 2, label: 'b', arrows:'to'},
     {from: 2, to: 3, label: 'c', arrows:'to'}
 ]);

 // provide the data in the vis format
```

[3] http://visjs.org/

```
var data = {
    nodes: nodes,
    edges: edges
};

// initialize the graph !
// We assume that divContainer and options are already initialized
var network = new vis.Network(divContainer, data, options);
```

The previous code produces the graph displayed in Figure 4.3.

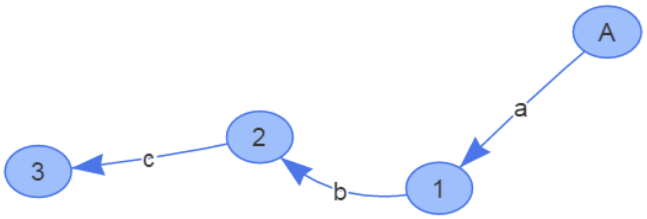


Figure 4.3: Graph: $A = a.b.c.\emptyset$

4.5.2 GraphTransition

The transitions produced earlier are not adapted for the graph visualization framework. A *Transition* is a shift between two internal states. Those internal states are sub-equations as you may remember from 4.4. In contrast datasets expect numerical id and all the transition between processes, not just the internal ones. Moreover, one may have to add some options like colors, arrow directions, etc.

To fill the gap, we will create a new object. This new object will be a transformation from transitions already generated. It is defined as follows:

```
case class GraphTransition(source: Int, action: String, target: Int,
    sourceLabel: Option[String]=None) extends
    AbstractTransition[Int,String]
```

Let us illustrate the link between *Transtion* and *GraphTransition* with the following equation $A = a.b.c.\emptyset$:

```
// A is defined as A = a.b.c.0
// A1 is defined as b.c.0
Transition(A, OutputAction(a), A1)
```

```
GraphTransition(0,"a",1, Some("A"))

//A2 is defined as c.0
Transition(A1, OutputAction(a), A2)
GraphTransition(1,"b",2)

//A3 is defined as 0
Transition(A2, OutputAction(a), A3)
GraphTransition(2,"c",3)
```

Given that we have illustrated the link between *Transition* and *GraphTransition*, we now move to describe the transformation between them.

To do so, we will process the set of transitions generated earlier. We will first produce the initial edges with the related nodes. Then, we will transform all the remaining transitions into *GraphTransitons*.

When a transition shifts from state to state, the reached state may be a reference to a process. If there is such a reference, we try to fetch its graph.

If it has not been generated yet, we generate it by a recursive call to *generateGraphTransition*. Then we link the first node of this sub-graph to an existing node of the graph being generated.

Of course, those referenced process can be recursive. This explains why we always generate the first transition along with the two nodes related as a first step.

Let's suppose our algorithm encountered a recursive process. When we try to fetch the first node of its sub-graph, which is actually the graph being processed, its first state may not have been generated yet. This can lead us to an infinite generation of graph. To avoid this issue, the first step in generating the graph is to create its first node.

```
def generateGraphTransitions(secondaryProcessTransition:
    List[Transition], primaryProcess: String, secondaryProcess:
    String): List[GraphTransition] = {

  def findEdgeSourceTarget(transit: Transition, nodeReferences:
      collection.mutable.ListMap[String, Int]): (Int, Int) = {...}

  def transformTransitionIntoTransitionGraph(transit: List[Transition],
      edges: List[GraphTransition]):List[GraphTransition] = {...}

//Link between equation and its numerical id.
```

```
    val nodeReferences = collection.mutable.ListMap[String, Int]()

  //The equation corresponding to the process reference
    val processDefinition = ContextHolder.formula.find(x =>
        x.getId().equals(secondaryProcess)).get

    val (firstTransitions, otherTransitions) =
        secondaryProcessTransition.partition(x =>
        (x.source.equals(processDefinition.getRight)))

  //Create the initial edges with the related nodes
  // A = a.b.c.0 + l.k.0
  //--> GraphTranstion(0,"a",1,"A") GraphTranstion(0,"l",2,"A")
    val initialEdges = generateInitialEdges(firstTransitions,
        nodeReferences, primaryProcess, secondaryProcess)

  // Transform all the remaining Transitions into GraphTransitions
    transformTransitIntoTransitGraph(otherTransitions, initialEdges)
  }
```

The algorithm proceeds as follows:

a) *nodeReferences* is created. This map uses process equation as a key and node id as a value;

b) *processDefinition* contains the current equation AST;

c) *processDefinition* allows us to extract the first transitions. They are then stored into $firstTransition$. The remaining transitions are stored into *otherTranstions*;

d) Each transition contained in $firstTransitions$ is transformed into a $GraphTransition$ and then stored into *initialEdges*. As explained earlier this helps us to avoid an infinite recursive loop;

e) Finally, each transition remaining in *otherTranstions* are transformed into $GraphTransition$ by the function $transformTransitIntoTransitGraph$. As explained previously, if during the generation a new process is found, then it recursively calls *generateGraphTransitions*.

The next step will be to glue the back-end and the front-end.

4.6 Web Framework

Scala allows integration of web frameworks running on the Java Virtual Machine (JVM). Our main criteria to select a web framework was a seamless interoperability with Scala. One name constantly arose: Play Framework.

4.6.1 Play Framework

Play is an open source web application framework. Play is written in Scala and can be used from Java or Scala. It was created in 2007 by Guillaume Bort [7]. Play follows the model–view–controller (MVC) architectural pattern[4].

Let us provide an overview of the Play framework, summarizing the contrasting views of Yevgeniy Brikman and Jonas Bonér [10, 23]. Yevgeniy Brikman is a former staff software developer at LinkedIn who led the project that moved LinkedIn's infrastructure to the Play Framework [5]. Jonas Bonér is the CTO of Lightbend, a commercial company founded by Martin Odersky, the creator of the Scala programming language and Jonas Bonér, the creator of the Akka middle-ware, and Paul Phillips [6].

Pros

- Interoperability with Scala and Java
- Functional programming: Play/Scala tries to stick to functional programming principles: immutable data, pure functions, no side effects.
- Developer friendly: Convention over configuration - Hot code reloading - Hit refresh work flow - Built in testing tools - IDE support
- Twirl give us Type-safe compiled templates where we can write directly Scala code.
- Reactive: Play is built on Netty and Akka[7], thus it supports non-blocking I/O.
- Open source

[4] `https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`
[5] `https://www.linkedin.com/in/jbrikman/`
[6] `https://www.lightbend.com/company/leadership`
[7] Akka is a concurrency framework based on the actor's model: `http://akka.io/`

- Error handling. In dev mode: compile and runtime errors are showed with a meaningful error message in the browser.

Cons

- Retro-compatibility : Major releases of Play are not retrocompatible. Therefore it may require rewriting some parts of the code base.
- Functional programming. The functional style can have drawbacks. For some problems a bit of mutable state is the best way to solve a problem. Moreover, some existing libraries that cannot be changed depends on mutable state, side effects, thread local. To work with those library with Play/Scala can be tricky. Scala has a steep learning curve.
- SBT is Play's build system. SBT is very powerful but to integrate SBT in a large code base the learning curve can be steep.

As mentioned in the pros, Play works seamlessly with Scala and is developer friendly. Because in our case, we are developing a prototype, all the integration issues are not relevant to us. We will reviewed the advantages and disadvantages of the play framework in the conclusion.

4.6.2 Converting GraphTranstion to JSON

Play provides us helper functions to convert Scala objects into JSON. The JSON is natively handled by JavaScript and transformed into JavaScript objects.

To gives the nodes and edges expected by *vis.js* we proceed as follows:

First we extract all the different node identifiers located in the GraphTransitions as source and target. This step is achieved by the function $TransitionHelper.getNode$. Then we convert the received nodes to JSON. This step is performed by the function $JsonHelper.getNodesJson$. Afterwards we convert the GraphTransition into JSON. This step is processed by $getEdgesJson$. GraphTransitions actually depicts the edge of our graph. Finally, we give the nodes and the edges in JSON format to the view via the OK function.

```
//Converting Scala GraphTransition into JSON object
def getEdgesJson(transition: List[GraphTransition]):JsObject ={
    implicit val transitionFormat = Json.format[GraphTransition]
    Json.obj("transition" -> transition)
   }

//We extract all different node IDs
```

```
val nodes = TransitionHelper.getNode(graphTransitions)

val nodesJson = JsonHelper.getNodesJson(nodes)
val edgesJson = JsonHelper.getEdgesJson(graphTransitions)

//JSON objects are sent to the view
Ok(edgesJson ++ nodesJson)
```

4.7 Limitations and Related work

4.7.1 Limitations

In section 4.4 Transitions, we detailed the behavior of the dispatcher 4.4. The dispatcher traverses the AST to build the internal transitions of CCS equations.

For each process matched while the function travels across the AST a sub-function is called to generate the related transitions. For instance the function $generateSummationTransition$ is called to build the transition for the + operator.

We also stated that the dispatcher does not handle $ReferencedProcess$ to avoid a recursive loop. But this last statement has a consequence on the different types of CCS equation covered by our application.

Indeed, at each side of an operator, an unguarded $ReferencedProcess$ previously defined can be assigned. Let us demonstrate this by the subsequent CCS equations:

$$A = a.b.\emptyset \tag{4.1}$$

$$B = c.d.\emptyset \tag{4.2}$$

$$C = A + B \tag{4.3}$$

$$D = A \mid B \tag{4.4}$$

$$E = A \tag{4.5}$$

Since, the generating sub-functions recursively call the dispatcher to generate the transitions, as in:

```
def generateSummationTransition(summationP:
    SummationProcess):List[Transition] = {
  //The dispatcher creates left side + right sided transitions
```

```
  dispatcher(summationP.left)... ++ dispatcher(summationP.right)...

}
```

Equations 4.3 to 4.5 are not managed by our tool.

To cover more cases in our software, a solution is to rewrite CCS equations and replace the unguarded processes by their definitions. In doing so, an equation such as $C = A + B$ becomes $C = a.b.\emptyset + c.d.\emptyset$ and is definitely processed by our tool.

The rewriting algorithm is called just before the parser. This choice permits us to create an AST without unguarded processes. The work-flow change in the following way:

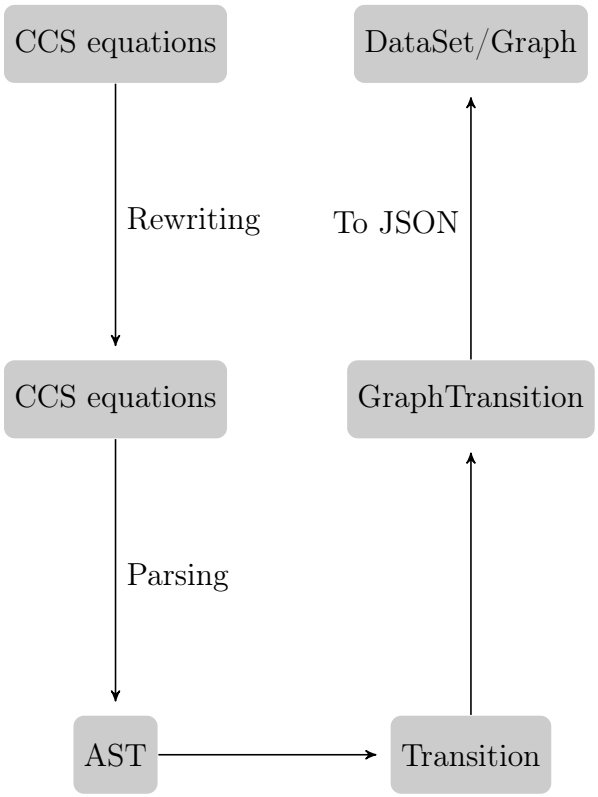


Figure 4.4: Implementation diagram revised

Let's have a look at the algorithm:

```
def toGuarded(allProcesses: Map[String, String]): Map[String, String] = {
```

```
    val p = allProcesses.map(s => s._1 ->
    unguardedProcessRegex
        .replaceAllIn(s._2,b=>replaceProcess(allProcesses,b.toString)))
    if (isGuarded(p))
      toGuarded(p)
    else
      p
  }
```

This function takes as an argument a Map containing all the CCS equations. The key is the formula id and the value is its definition e.g.: $A \longrightarrow a.b.\emptyset$. The first step is to replace all the unguarded processes by their definitions. If there are several level of indirections, the rewritten equations can be still unguarded. The function will reprocess the new equations until there are no unguarded processes.

$$A = D \quad D = E \quad E = e.d.\emptyset$$

Before processing	First Iteration	Last Iteration
$A \rightarrow D$	$A \rightarrow E$	$A \rightarrow e.d.\emptyset$
$D \rightarrow E$	$D \rightarrow e.d.\emptyset$	$D \rightarrow e.d.\emptyset$
$E \rightarrow e.d.\emptyset$	$E \rightarrow e.d.\emptyset$	$E \rightarrow e.d.\emptyset$

Table 4.1: *toGuarded* example

The results of this approach were still unsatisfactory for advanced cases. However, unguarded recursive equation such as $A = a.b.(A + l.a)$ are not managed by our tool.

Moreover, for the sake of simplicity the | only covers the basic case: $A = C \mid D$ where C and D are defined as terminal CCS equations.

Actually, the | generating transition function does not make recursive calls to the dispatcher. It could be modified to handle as much cases as the + operator. But this modification would imply to add extra complexity to the code. Furthermore, it would be tedious to prove that all the possibilities of the CCS language are managed.

Those pitfalls can be avoided with a different approach. One of them is described in the next subsection.

4.7.2 Related work

Numerous CCS workbenches have been previously developed. A non exhaustive list can be found in the companion web site of the book Reactive Systems (Aceto et al., 2007) [4].

One of them caught our attention because it is a modern web based tool written in TypeScript that enables graph display: CAAL (Concurrency Workbench, Aalborg Edition). It consists of "a web-based tool for modelling and verification of concurrent processes. The tool is primarily designed for educational purposes and it supports the classical process algebra CCS together with its timed extension TCCS.

It allows one to compare processes with respect to a range of strong/weak and timed/untimed equivalences and preorders (bisimulation, simulation and traces) and supports model checking of CCS/TCCS processes against recursively defined formulae of Hennessy-Milner logic.

The tool offers a graphical visualizer for displaying labelled transition systems, including their minimization up to strong/weak bisimulation, and process behaviour can be examined by playing (bi)simulation and model checking games or via the generation of distinguishing formulae for non-equivalent processes."[2].

The source code of CAAL is available at the following url: `https://github.com/CAAL/CAAL`

To better grasp the tool, let us give more details of the approach tailored by CAAL. During the parsing stage, for each encountered process, transitions are produced on the fly. In order to do this, the process is sliced into sub-process/sub-states as in: $a.b.\emptyset \longrightarrow b.\emptyset \longrightarrow \emptyset$.

The process preserves a reference to its sub-process(es). The union of the transitions of each sub-process allows it to acquire the whole set of the process transitions.

The set of transitions for each process/sub-process are maintained in a map with the corresponding ID.

By having a map with all sub-process transitions and a link between processes and sub-processes we could simplify our software. In this manner, the transitions of reference to a process in a formula can be treated as a straightforward call to their identifiers in the map. Cases such as $A + B$, $A \mid B$, and $a.b.c(A + (A \mid B))$ are treated effortlessly.

We think that such an approach could permit to cover all the cases of the CCS

language without limitations or extra complexity. To achieve that, our work should be refactored to implement that design.

Nevertheless, it is worth pointing out here that our work presents the originality of building a tool from scratch in using a three tier architecture: a back-end layer has the modeling language based on Scala, a front-end layer built on a web client based on JavaScript and vis.js, and a third layer with play as a web framework to link the front-end to the back-end Scala-based processing tool.

5 Behavioral Equivalences

In the introduction of the CCS language we said that CCS can be used to describe both the specification and the implementation of a process. We also claimed that the notion of behavioral equivalence can be used to check if an implementation satisfies a specification 2.7. To an external observer, in the case in which an implementation satisfies a specification, both of them will have the same behavior and will be completely similar.

Obviously, the specification and the implementation are not at the same level of abstraction and the two formulas are not exactly the same. Therefore, in this notion of equivalence, some parameters have to be taken into account and others not.

In the next pages, we try to find a suitable notion of equivalence. We will first define the expected criteria of such a notion. Then we will explore different tracks and implement them into our tool.

5.1 Criteria for a good behavioral equivalence

Since Chapter 2 on CCS, we are using the term of *equivalence*. It is now time to give its mathematical definition. Quoting verbatim from Reactive Systems [1, p. 32] we define *equivalence* as follows:

Let X be a set. A binary relation over X is a subset of $X \times X$, the set of pairs of elements of X. If R is a binary relation over X, we often write xRy instead of $(x, y) \in R$.

An equivalence relation over X is a binary relation R that satisfies the following constraints:

- R is reflexive i.e. xRx for each $x \in X$;
- R is symmetric i.e. xRy implies yRx, for all $x, y \in X$;
- R is transitive i.e. xRy and yRz imply xRz, for all $x, y, z \in X$.

A reflexive and transitive relation is a preorder [1, p. 32].

Let us review each property of this definition:

- $Reflexive$: we expect each process to be the correct implementation of itself, so the relation should be reflexive.
- $Transitive$: if we want to derive step by step the implementation from the specification and preserve the behavior, the relation should be transitive. In this case, we can start from the specification and convert it into an intermediate stage until we reach the implementation.

$$Spec = Spec_0 \ R \ Spec_1 \ R \ Spec_2 \ R \ ... \ R \ Spec_n \ = \ Imp$$

$$Spec \ R \ Imp$$

- $Symmetric$: we expect that Imp behaves like Spec and vice-versa, so the relation should be symmetric [1, p. 33].

We said earlier that the specification and the implementation are not at the same level of abstraction. A desirable feature would be to use the implementation or the specification interchangeably as a part of larger system without affecting the general behavior. This feature is called the *congruence*.

The last feature that we expect is an *observational equivalence* of the behavior. We want this equivalence to be based on what an observer can see rather than on the number of transitions between states, the state's name or the process structure.

5.2 Trace equivalence

As we said in the section 2.2, LTS models process states and transitions between them. The shifts between states are triggered by actions.

In the light of the details offered by this view, a number of different notions of equivalence have been proposed. One of them comes directly from the classic theory of automata, the *trace* [1, p. 34].

The *trace* of a process P is a sequence of actions that results from a maximal path of transitions.

$$P \;=\; P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_{n-1}} P_{n-1} \xrightarrow{\alpha_n} P_n$$

where $\alpha_i \in Act$ and $n \geq 0$.

$Traces(P)$ is the collection of all the traces of P. $Traces(P)$ depicts all the sequence of actions available in our process.

Let us consider two processes A and B. If a sequence of actions is present in the traces of A but it is not present in the traces of B, then we can conclude that they are not equivalent.

This view is totally legitimate for deterministic automata. But what happens for reactive systems such as the two vending machines described in here below ?

$$VM \stackrel{def}{=} coin.(\overline{chocolate}.VM + \overline{yogurt}.VM) \tag{5.1}$$

$$VM' \stackrel{def}{=} coin.\overline{chocolate}.VM' + coin.\overline{yogurt}.VM' \tag{5.2}$$

Both machines can provide chocolate and yogurt. They have the same traces as well:

$$\begin{aligned} Traces(VM) =& \{coin, \overline{chocolate}\} \cup \\ & \{coin, \overline{yogurt}\} \end{aligned} \tag{5.3}$$

$$\begin{aligned} Traces(VM') =& \{coin, \overline{chocolate}\} \cup \\ & \{coin, \overline{yogurt}\} \end{aligned} \tag{5.4}$$

Let's define a chocolate addict user to interact with our machines.

$$CU \stackrel{def}{=} \overline{coin}.chocolate.CU$$

Consider now the two next equations:

$$(VM \mid CU) \setminus \{coin, chocolate, yogurt\} \tag{5.5}$$

$$(VM' \mid CU) \setminus \{coin, chocolate, yogurt\} \tag{5.6}$$

The restriction permits us to force the communication between the chocolate addict and the machine. By using the SOS rules, one can notice that VM performs infinitely many τ transitions.

In contrast, VM' can reach a deadlock state:

$$(chocolate.CU \mid \overline{yogurt}.VM')$$

The chocolate addict is expecting to receive some chocolates to continue to operate properly, and the machine is only able to deliver yogurt.

Therefore we need another notion to tell apart two systems having the same traces but displaying a different reactive behavior.

5.3 Simulation & Bisimulation

5.3.1 Theoretical notions

Simulation

Another relation available on LTS is the simulation relation. Intuitively, a system A simulates a system B, if A can match all of the transitions of B.

Let's give first its formal definition [11, p. 186]:

Let $T = (S, A, \longrightarrow)$ be a transition system. A binary relation $R \subseteq S$ x S is a *simulation* if whenever $(s, t) \in R$ and α is action:

if $s \xrightarrow{\alpha} s'$, then $t \xrightarrow{\alpha} t$' for some t such that s' R t'.

We write a simulation relation between the process A and B, as $B \preceq^S A$. This should be understood as A simulates B. Let's notice that the relation $\preceq^S$ is a preorder.

Let's now define the *simulation equivalence* as:

$$A \simeq^S B \Rightarrow A \preceq^S B \wedge B \preceq^S A$$

This means that B simulates A and A simulates B.

Let's turn now to our two vending machines and see how they react to this new notions.

First, let's see if VM can simulate VM', $VM' \preceq^S VM$:

To do so, we first write the intermediate states for each machine:

$$VM \begin{cases} VM \stackrel{def}{=} coin.(\overline{chocolate}.VM + \overline{yogurt}.VM) \\ VM1 \stackrel{def}{=} \overline{chocolate}.VM + \overline{yogurt}.VM \end{cases}$$

$$VM' \begin{cases} VM' \stackrel{def}{=} coin.\overline{chocolate}.VM' + coin.\overline{yogurt}.VM' \\ VM1' \stackrel{def}{=} \overline{chocolate}.VM' \\ VM2' \stackrel{def}{=} \overline{yogurt}.VM' \end{cases}$$

We define R, our simulation relation as :

$$R = \{(VM', VM,), (VM1', VM1), (VM2', VM1)\}$$

The relation R is portrayed in Figure 5.1.

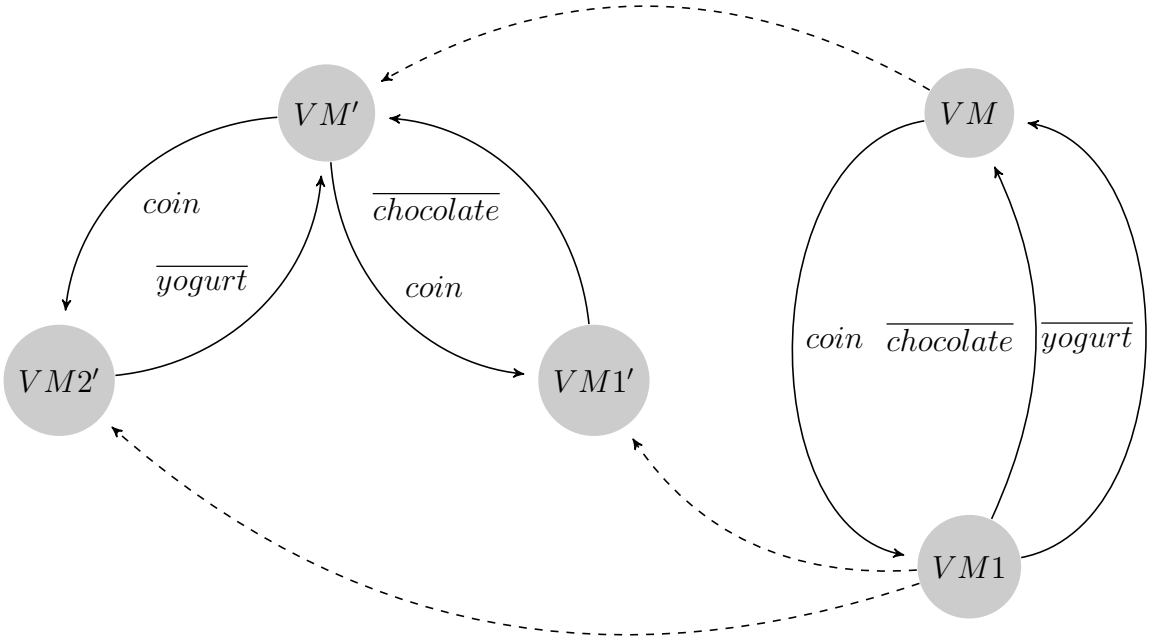


Figure 5.1: $VM' \preceq^S VM$

- Let's consider the first pair (VM', VM)

  - $VM' \xrightarrow{coin} VM1'$ is matched by $VM \xrightarrow{coin} VM1$.
  - $VM' \xrightarrow{coin} VM2'$ is matched by $VM \xrightarrow{coin} VM1$.
- Then the pair $(VM1', VM1)$
  - $VM1' \xrightarrow{\overline{chocolate}} VM'$ is matched by $VM1 \xrightarrow{\overline{chocolate}} VM$.
- Finally the pair $(VM2', VM1)$
  - $VM2' \xrightarrow{\overline{yogurt}} VM'$ is matched by $VM1 \xrightarrow{\overline{yogurt}} VM$.

That concludes the proof that $VM' \preceq^S VM$.

Let's see if the other way around is also true and $VM \preceq^S VM'$ holds.

- Let's consider the first pair (VM, VM')
  - $VM \xrightarrow{coin} VM1$ can be matched by $VM' \xrightarrow{coin} VM1'$.
  - $VM \xrightarrow{coin} VM1$ can also be matched by $VM' \xrightarrow{coin} VM2'$ but finding a transition is sufficient.
- Then the pair $(VM1, VM1')$
  - $VM1 \xrightarrow{\overline{chocolate}} VM$ is matched by $VM1' \xrightarrow{\overline{chocolate}} VM$'.
  - $VM1 \xrightarrow{\overline{yogurt}} VM$ cannot be matched by any transitions belonging to $VM1'$.
- Finding one non matching transition is enough but we examine the pair $(VM2', VM1)$ to be exhaustive
  - $VM1 \xrightarrow{\overline{yogurt}} VM$ is matched by $VM2' \xrightarrow{\overline{yogurt}} VM'$.
  - $VM1 \xrightarrow{\overline{chocolate}} VM$ cannot be matched by any transitions belonging to $VM2'$.

Therefore, $VM \simeq^S VM'$ does not hold.

This example emphasizes the fact that we are forced to reject the rule:

$$\alpha.(A + B) = \alpha.A + \alpha.B$$

At first glance, the notion of *simulation equivalence* seems to fit our requirements for equivalence behavior. But let's now consider the following vending machines :

$$CVM \stackrel{def}{=} coin.\overline{chocolate}.\emptyset \tag{5.7}$$

$$EVM \stackrel{def}{=} coin.\overline{chocolate}.\emptyset + coin.\emptyset \tag{5.8}$$

The CVM (Chocolate Vending Machine) delivers a bar of chocolate after receiving a coin whereas EVM (Evil Vending Machine) has a more erratic behavior. EVM can deliver a chocolate bar or just keep the coin.

Let us examine those two machines with our notion of *simulation equivalence*. First let's define the sub-states for each equation and then the simulation relation for each *simulation preorder* as illustrated in the Figure 5.2.

$$CVM \stackrel{def}{=} coin.CVM1$$

$$CVM1 \stackrel{def}{=} \overline{chocolate}.CVM2$$

$$CVM2 \stackrel{def}{=} \emptyset$$

$$EVM \stackrel{def}{=} coin.EVM1 + coin.EVM2$$

$$EVM1 \stackrel{def}{=} \overline{chocolate}.EVM3$$

$$EVM2 \stackrel{def}{=} \emptyset$$

$$EVM3 \stackrel{def}{=} \emptyset$$

$$R_1 = \{(EVM, CVM),\ (EVM1, CVM1),\ (EVM2, CVM1),\ (EVM3, CVM2)\}$$

$$R_2 = \{(CVM, EVM),\ (CVM1, EVM1),\ (CVM2, EVM3)\}$$

$R_1 : EVM \preceq^S CVM$

- The pair (EVM, CVM)
  - $EVM \stackrel{coin}{\longrightarrow} EVM1$ is matched by $CVM \stackrel{coin}{\longrightarrow} CVM1$.
  - $EVM \stackrel{coin}{\longrightarrow} EVM2$ is matched by $CVM \stackrel{coin}{\longrightarrow} CVM1$.
- The pair $(EVM1, CVM1)$
  - $EVM1 \stackrel{\overline{chocolate}}{\longrightarrow} EVM3$ can be matched by $CVM1 \stackrel{\overline{chocolate}}{\longrightarrow} CVM2$.
- The pair $(EVM2, CVM1)$

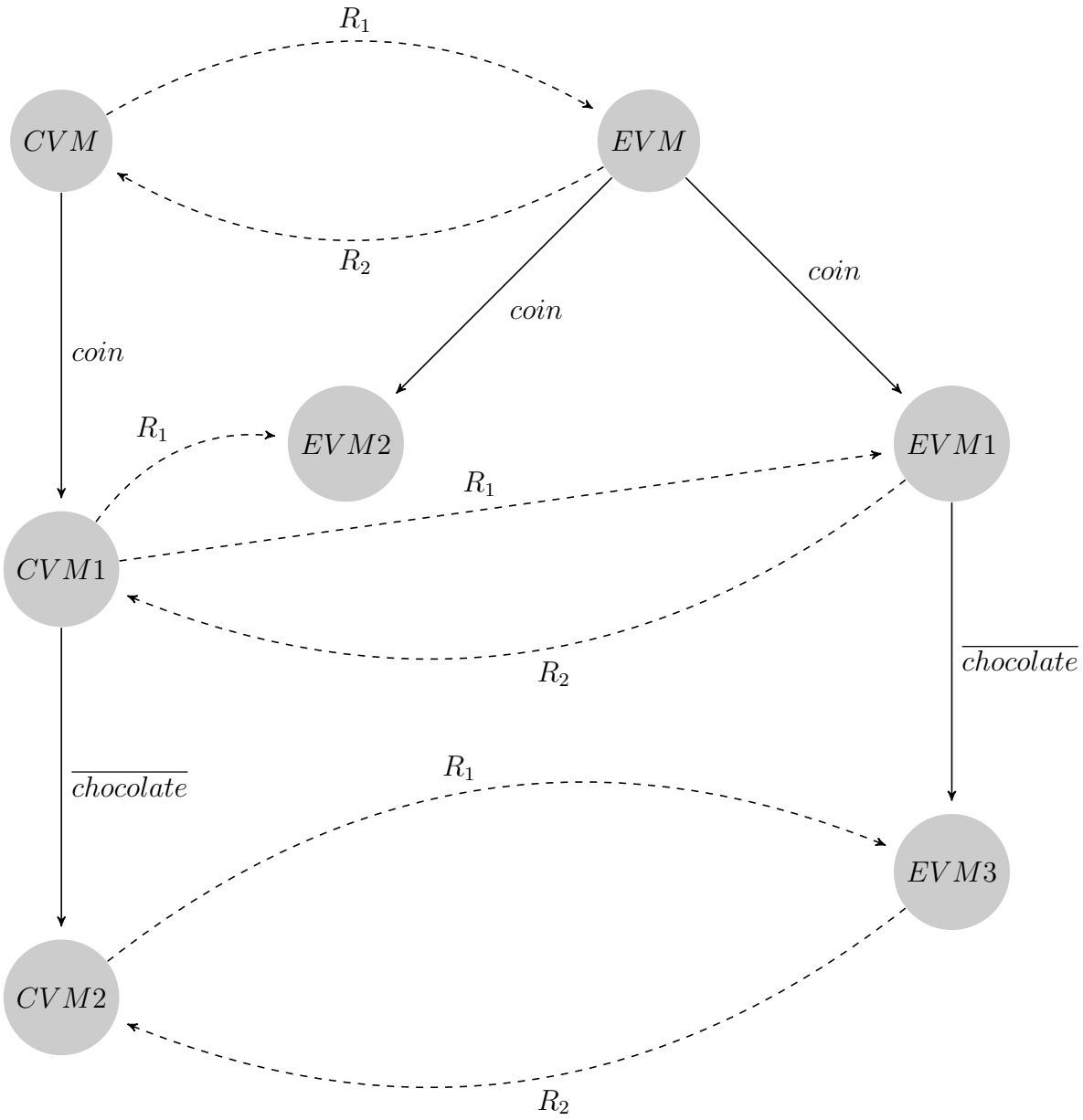


Figure 5.2: $CVM \simeq^S EVM$

  - $EVM2$ has no outgoing transition and thus can be trivially matched by $CVM1$.

- The pair $(EVM3, CVM2)$
  - $EVM3$ has no outgoing transition and thus can be trivially matched by $CVM2$.

$R_2 : CVM \preceq^S EVM$

- The pair (CVM, EVM)
  - $VM \xrightarrow{coin} CVM1$ can be matched by $EVM \xrightarrow{coin} EVM1$ or $EVM \xrightarrow{coin} EVM2$.
- The pair$(CVM1, EVM1)$
  - $CVM1 \xrightarrow{\overline{chocolate}} CVM2$ is matched by $VM2' \xrightarrow{\overline{chocolate}} VM'$.
- Finally, the pair$(CVM2, EVM3)$
  - $CVM2$ has no outgoing transition and thus can be trivially matched by $EVM3$.

The evil automaton can simulate the good one and vice versa. Therefore this notion of equivalence has been proven inadequate for our requirements. This pitfall can be avoided by the bisimulation discussed in the next section.

Strong Bisimulation

In the above discussion in simulation 5.3.1, we noticed that a behavioral notion of equivalence should allow us to separate two processes with different deadlock potentials.

To be equivalent, they should not only be able to simulate the other process globally, but each state should afford the same transition that its mirror state. To ensure the previous requirements hold, a symmetric *simulation*, called *strong bisimulation*, has been proposed by David Park [1, p. 37].

Let $T = (S, A, \longrightarrow)$ be a transition system. A binary relation $R \subseteq S$ x S is a *strong bisimulation* if whenever $(s, t) \in R$ and α is action:

if $s \xrightarrow{\alpha} s'$, then $t \xrightarrow{\alpha} t$' for some t such that s' R t';

if $t \xrightarrow{\alpha} t'$, then $s \xrightarrow{\alpha} s$' for some s such that s' R t'.

In order to understand this definition it is beneficial to consider the following example:

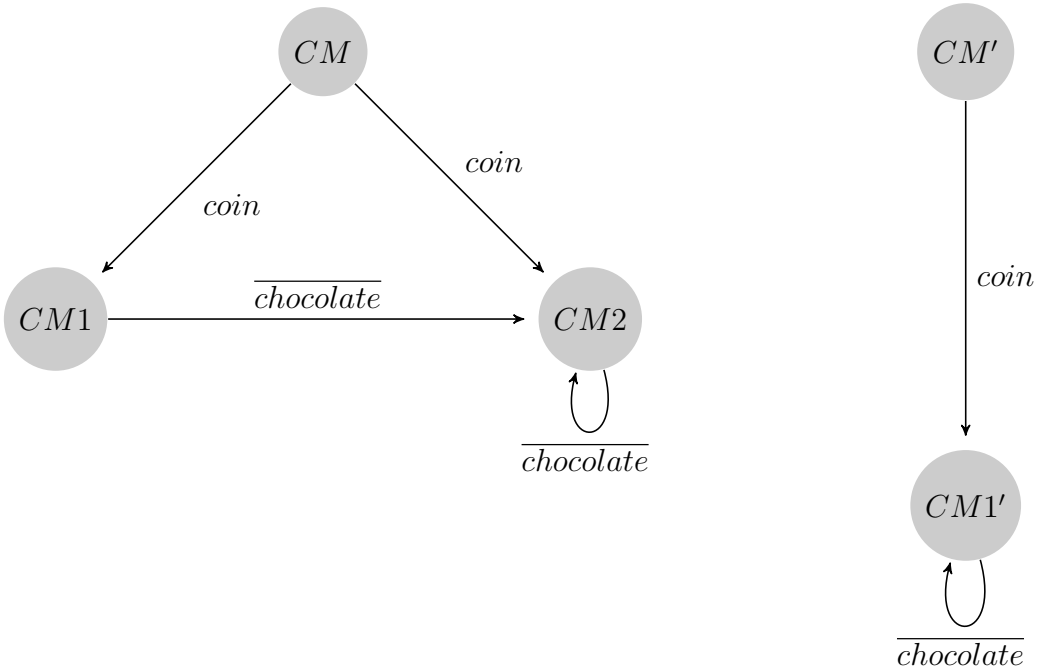


Figure 5.3: $CM \simeq^B CM'$

Intuitively, once a coin is inserted both machines give an infinite amount of chocolate. Let's examine them formally and prove that there is indeed a strong bisimulation between them.

We define R as follows :

$$R = \{(CM, CM')(CM1, CM1')(CM2, CM1')\}$$

- Let us consider the first pair (CM, CM')
  - From CM:
    - $CM \xrightarrow{coin} CM1$ is matched by $CM' \xrightarrow{coin} CM1'$ and $(CM1, CM1') \in R$;
    - $CM \xrightarrow{coin} CM2$ is matched by $CM' \xrightarrow{coin} CM1'$ and $(CM1, CM1') \in R$.
  - From CM':

  - $CM' \xrightarrow{coin} CM1'$ is matched by $CM \xrightarrow{coin} CM1$ and $(CM1, CM1') \in R$ (finding one matching transition is sufficient).

- Then the pair$(CM1, CM1')$
  - From $CM1$:
    - $CM1 \xrightarrow{\overline{chocolate}} CM2$ is matched by $CM1' \xrightarrow{\overline{chocolate}} CM1'$ and $(CM2, CM1') \in R$.
  - From $CM1'$:
    - $CM1' \xrightarrow{\overline{chocolate}} CM1'$ is matched by $CM1 \xrightarrow{\overline{chocolate}} CM2$ and $(CM2, CM1') \in R$.
- Finally, the pair$(CM2, CM1')$
  - From $CM2$:
    - $CM2 \xrightarrow{\overline{chocolate}} CM2$ is matched by $CM1' \xrightarrow{\overline{chocolate}} CM1'$ and $(CM2, CM1') \in R$.
  - From $CM1'$:
    - $CM1' \xrightarrow{\overline{chocolate}} CM1'$ is matched by $CM2 \xrightarrow{\overline{chocolate}} CM2$ and $(CM2, CM1') \in R$.

This ends the proof that R is bisimulation.

Let's turn back to CVM and EVM and see how they react to bisimulation. We define R as follows:

$$R = \{(EVM, CVM),\ (EVM1, CVM1),\ (EVM2, CVM1),\ (EVM3, CVM2)\}$$

- The pair (EVM, CVM)
  - From EVM:
    - $EVM \xrightarrow{coin} EVM1$ is matched by $CVM \xrightarrow{coin} CVM1$ and $(EVM1, CVM1) \in R$;
    - $EVM \xrightarrow{coin} EVM2$ is matched by $CVM \xrightarrow{coin} CVM1$ and $(EVM2, CVM1) \in R1$
  - From CVM:
    - $CVM \xrightarrow{coin} CVM1$ is matched by $EVM \xrightarrow{coin} EVM1$ and $(EVM1, CVM1) \in R$ (finding one matching transition is sufficient).

- The pair $(EVM1, CVM1)$
  - From $EVM1$:
    - $EVM1 \xrightarrow{\overline{chocolate}} EVM3$ is matched by $CVM1 \xrightarrow{\overline{chocolate}} CVM2$ and $(EVM3, CVM2) \in R$.
  - From $CVM1$:
    - $CVM1 \xrightarrow{\overline{chocolate}} CVM2$ is matched by $EVM1 \xrightarrow{\overline{chocolate}} EVM3$ and $(EVM3, CVM2) \in R$.
- The pair $(EVM2, CVM1)$
  - From $EVM2$:
    - $EVM2$ has no outgoing transition and thus can be trivially matched by $CVM1$. and $(EVM2, CVM1) \in R$.
  - From $CVM1$:
    - $CVM1 \xrightarrow{\overline{chocolate}} CVM2$ cannot be matched by any transition from $EVM2$.

The last step is sufficient to conclude that $CVM \simeq^B EVM$ is false.

5.3.2 Algorithms

Given that, the theoretical bases of simulation and bisimulation have been introduced, let us discover how they can be turned into a sequence of actions expressed in an algorithm. The simulation and bisimulation algorithms will be implemented afterwards in our tool.

The general idea of the (bi)simulation algorithm is to combine two CCS equations into a set of tuples. Those tuples are formed with states that afford the same actions. Afterwards, a control function is applied on the set to remove tuples that are not a part of the (bi)simulation relation. This function is applied as long as the set is changing. Once the function reached a fixed point, the elements within the set depict the greatest (bi)simulation relation.

Simulation preorder

Let us consider the two following LTS, $(S_1, A_1, \longrightarrow)$ and $(S_2, A_2, \longrightarrow)$. We create the initial set of states H_0 with the tuple S_1 and S_2 if they afford the same action

at the same level in the path of actions.

$$(S_1, S_2) \in H_0 \iff A_1(S_1) = A_2(S_2)$$

From the previous set we will create the following ones:

$$(S_1, S_2) \in H_{i+1} \iff \begin{cases} (S_1, S_2) \in H_i \wedge \\ \forall\ S_1' \in S_1.\ S_1 \xrightarrow{\alpha} S_1'\ \exists\ S_2' \in S_2.\ S_2 \xrightarrow{\alpha} S_2' \wedge \\ (S_1', S_2') \in H_i \end{cases}$$

$$Until\ H_{i+1} = H_i$$

Step two can be decomposed into four sub-steps.

(a) The pair (S_1, S_2) should be present in the previous set.

(b) For each transition afforded by S_1 at least one matching transition should start from S_2.

(c) The new tuple of target states (S_1', S_2') should belong to the previous set (H_i).

(d) We repeat (a), (b) and (c) for each state presents in the previous set. Each time the conditions are met the tuple is added into H_{i+1}. The algorithm stops when the previous set equals the current set or the set of states is empty.

Finally, the last set of states is evaluated. If the root elements are still there, the algorithm returns true. Otherwise it returns false.

Let's see how our two vending machines VM and VM' react to this algorithm.

$$VM \preceq^S VM' \begin{cases} H_O = \{(VM', VM,), (VM1', VM1), (VM2', VM1)\} \\ H_1 = \{(VM', VM,)\} \\ H_2 = \{\emptyset\} \\ H_3 = H_2 \end{cases}$$

As $(VM', VM,) \notin H_3$, $False$ is returned

$$VM' \preceq^S VM \begin{cases} H_O = \{(VM', VM,), (VM1', VM1), (VM2', VM1)\} \\ H_1 = \{(VM', VM,), (VM1', VM1), (VM2', VM1)\} \\ H_1 = H_0 \end{cases}$$

As $(VM', VM,) \in H_1$, $True$ is returned

Strong bisimulation

To meet the bisimulation definition, some extra constraints have to be added to the *simulation* algorithm :

$$(S_1, S_2) \in H_0 \iff A_1(S_1) = A_2(S_2)$$

$$(S_1, S_2) \in H_{i+1} \iff \begin{cases} (S_1, S_2) \in H_i \wedge \\ \forall\ S_1' \in S_1.\ S_1 \xrightarrow{\alpha} S_1'\ \exists\ S_2' \in S_2.\ S_2 \xrightarrow{\alpha} S_2' \wedge \\ (S_1', S_2') \in H_i \\ \forall\ S_2' \in S_2.\ S_2 \xrightarrow{\alpha} S_2'\ \exists\ S_1' \in S_1.\ S_1 \xrightarrow{\alpha} S_1' \wedge \\ (S_2', S_1') \in H_i \end{cases}$$

$$Until\ H_{i+1} = H_i$$

The computation of the initial set has not changed. But, once the iteration on each tuple of states starts, the check becomes symmetric.

To do so, once the transitions from the first state of the tuple are checked by: $\forall\ S_1' \in S_1.\ S_1 \xrightarrow{\alpha} S_1'\ \exists\ S_2' \in S_2.\ S_2 \xrightarrow{\alpha} S_2'$, then the transitions starting from the second element of the tuple are verified by the added rule: $\forall\ S_2' \in S_2.\ S_2 \xrightarrow{\alpha} S_2'\ \exists\ S_1' \in S_1.\ S_1 \xrightarrow{\alpha} S_1'$.

Fixed points algorithms

Simulation and bisimulation algorithms are an application of Tarski's fixed point theorem. Tarski states that a monotonic function applied on complete lattice has a least and a largest fixed point [1, p. 80][22]. In our case H_0 is the top element of our complete lattice $Proc$ x $Proc$, and the last iteration of H_i is the largest fixed point of the (bi)simulation function applied to that set[1, p. 85-86].

The time complexity of both algorithms is in $\mathcal{O}(mn)$ for a labeled transition system with m transitions and n states. A more efficient algorithm, not discussed in this work, has been proposed by Paige and Tarjan in $\mathcal{O}(m\ log\ n)$ time [6].

5.3.3 Implementations

Simulation preorder

Let us turn to the implementation of the simulation preorder algorithm. The function $equivalenceVerification$, is a higher-order function used for *simulation* and *bisimulation* equivalence.

The behavior of this function depends on the functions given as arguments. The given arguments are:

- graphA: contains the graph transitions for the first equation.
- graphB: contains the graph transitions for the second equation.
- idA: the first equation name.
- idB: the second equation name.
- checkTransitionAlgorithm: verifies if $(S_1, S_2) \in H_{i+1}$.
- preCheck: verifies if there is a sufficient number of states in H_0.

```
def equivalenceVerification(graphA: List[GraphTransition], graphB:
    List[GraphTransition], idA: String, idB: String,
                            checkTransitionAlgorithm:
                                (List[GraphTransition],
                                List[GraphTransition], Set[(Int, Int)]) =>
                                Boolean,
                            preCheck: (List[(GraphTransition,
                                GraphTransition)], List[GraphTransition],
                                List[GraphTransition]) => Boolean):
                                Boolean = {

    val rootElementA = TransitionUtil.firstSource(graphA, Some(idA))
    val rootElementB = TransitionUtil.firstSource(graphB, Some(idB))

    val (initialStateSet, initialSet) = createInitialStateSet(graphA,
        graphB, Set(rootElementA), Set(rootElementB))

    if (!preCheck(initialSet, graphA, graphB))
      false
    else
      containRootElements(fixedPoint(graphA, graphB, initialStateSet,
          checkTransitionAlgorithm), rootElementA, rootElementB)
  }
```

equivalenceVerification verifies the simulation preorder as follows:

1. The initial states for each equation are retrieved
2. Creation of H_0 by *createInitialStateSet*
3. Check whether H_0 contains all the states present in *graphA*
4. *fixedPoint* iterates over H_i until it reaches a fixed point
5. *containRootElements* verifies whether the root elements are still in the *fixedPoint* result set.

Let's describe the H_0 production:

```
def createInitialStateSet(graphA: List[GraphTransition], graphB:
    List[GraphTransition], sourceElementsA: Set[Int], sourceElementsB:
    Set[Int]):(Set[(Int, Int)],List[(GraphTransition,
    GraphTransition)]) = {
  val initialSet = createInitialTransitionSet(graphA, graphB,
      sourceElementsA, sourceElementsB)
  val initialSetProcessSrc = initialSet.map(x => (x._1.source,
      x._2.source)).toSet
  val initialSetProcessTar = initialSet.map(x => (x._1.target,
      x._2.target)).toSet
  (initialSetProcessSrc ++ initialSetProcessTar, initialSet)
}
```

1. *initialSet* contains the tuples of matching transitions from each graph.
2. A map function is applied on the *initalSet* to transform the tuples of transitions into tuples of states. A first time for the source states and a second time for the transition target states.
3. Finally, the tuples of states and transitions are returned.

The first step of this function calls *createInitialTransitionSet* to gather all the matching transition for each graph. As intended by the code here below:

```
@tailrec
def createInitialSet(graphA: List[GraphTransition], graphB:
    List[GraphTransition], sourceElementAs: Set[Int], sourceElementsB:
    Set[Int], acc: Set[(GraphTransition, GraphTransition)]=Set()):
    Set[(GraphTransition, GraphTransition)] = {
```

```
    if (sourceElementAs.size == 0 || sourceElementsB.size == 0)
      acc
    else {
      val graphATransitions =
          findGraphTransitionBySources(sourceElementAs, graphA)
      val graphBTransitions =
          findGraphTransitionBySources(sourceElementsB, graphB)
      val tupleGraphs = createTuples(graphATransitions, graphBTransitions)
      val nextLevels = getNextLevels(tupleGraphs)
      if (acc.size == (tupleGraphs.toSet ++ acc).size)
        return acc
      createInitialSet(graphA, graphB, nextLevels._1, nextLevels._2,
          tupleGraphs.toSet ++ acc)
    }
  }
```

This function is tail recursive, as specified by the @*tailrec* annotation. The base case is reached once all the transitions for one of the graphs are consumed by the function. The inductive case traverses both graphs, level by level and processes as follows:

1. $findGraphTransitionBySources$ is called twice to retrieve transitions on the same level on both graphs from the states given in arguments.
2. $createTuples$ filters the same level transitions by matching action's name.
3. $getNextLevels$ fetches the next states.
4. If the graph is recursive, the base case will never be reached. To avoid infinite loop the previous set size is compared to the current set size.
5. Finally, $createInitialSet$ is called recursively with the next states. The new tuples of transitions are added to the accumulator.

Once the initial set is created, the algorithm inspects whether all states in $graphA$ are part of H_O.

```
  def preCheckSimulation(initialSet: List[(GraphTransition,
      GraphTransition)], graphA: List[GraphTransition], graphB :
      List[GraphTransition]): Boolean = {
    checkIfAllStatesPresentInInit(initialSet, graphA)
  }

    def checkIfAllStatesPresentInInit(initialSet: List[(GraphTransition,
        GraphTransition)], graphA: List[GraphTransition]):Boolean =
```

```
      initialSet.map(_._1.action).toSet == (graphA.map(_.action)).toSet
```

The astute reader may notice that $graphB$ is not used. The reason is to keep $equivalenceVerification$ as generic as possible in order to reuse it for $bisimulation$.

The next step is to iterate over the set of states until it reached a fixed point. This function takes a verification function, $checkAlgorithm$, as an argument. $checkAlgorithm$ tells whether a tuple can be added to H_i.

```
@tailrec
def fixedPoint(graphA: List[GraphTransition], graphB:
    List[GraphTransition], previousSet: Set[(Int, Int)],
                checkAlgorithm: (List[GraphTransition],
                    List[GraphTransition], Set[(Int, Int)]) =>
                    Boolean):Set[(Int, Int)] = {

    val currentSet = previousSet.flatMap(s => {
      val nextTransitionsA = getNextTransitions(s._1, graphA)
      val nextTransitionsB = getNextTransitions(s._2, graphB)

      if (checkAlgorithm(nextTransitionsA,nextTransitionsB,previousSet))
        Some(s)
      else
        None
    })

    if (previousSet.size == currentSet.size)
      currentSet
    else {
      fixedPoint(graphA, graphB, currentSet, checkAlgorithms)
    }
  }
}
```

This function is tail recursive. The base case is reached once the set of states stabilizes. In the inductive case the next transitions of each graph are collected. Then the $checkAlgorithm$ is applied and returns a boolean. Depending on the result an optional tuple is returned.

Finally, let's examine how the check algorithm is implemented for the simulation.

```
  def simulationCheck(nextTransitionsA:List[GraphTransition],
      nextTransitionsB:List[GraphTransition], previousSet: Set[(Int,
```

```
    Int)]):Boolean = {
  nextTransitionsA.forall(transit=> {
    val maybe_transition_b = nextTransitionsB.find(b =>
       b.action.equals(transit.action))
    maybe_transition_b match {
      case Some(graph) => existInPreviousSet(transit, graph,
         previousSet)
      case None => false
    }
  })
}
```

This function verifies if $\forall\ S'_1 \in S_1.\ S_1 \xrightarrow{\alpha} S'_1\ \exists\ S'_2 \in S_2.\ S_2 \xrightarrow{\alpha} S'_2\ \wedge$ $(S'_1, S'_2) \in H_i$. Indeed, for all transitions of $graphA$, the function tries to find an existing matching transition in $graphB$. If so, the function checks whether the tuples already belong to the previous set.

Simulation equivalence

The simulation equivalence is merely a simulation applied twice, one time in each direction. The implementation stems directly from this definition.

```
def simulationEquivalence(graphA: Set[GraphTransition], graphB:
    Set[GraphTransition], idA: String, idB: String,
                          checkTransitionAlgorithm:
                              (Set[GraphTransition],
                              Set[GraphTransition], Set[(Int, Int)]) =>
                              Boolean,
                          preCheck: (Set[(GraphTransition,
                              GraphTransition)], Set[GraphTransition],
                              Set[GraphTransition]) => Boolean): Boolean
                              = {
  Equivalence.equivalenceVerification(graphA, graphB, idA, idB,
     Equivalence.simulationCheck, Equivalence.preCheckSimulation) &&
    Equivalence.equivalenceVerification(graphB, graphA, idB, idA,
       Equivalence.simulationCheck, Equivalence.preCheckSimulation)
}
```

$equivalenceVerification$ is called twice with $simulationCheck$. The graphs passed as arguments are inverted between the two calls. A logical and ($\wedge$) is applied between the results of each call.

Strong Bisimulation

As explained in Section 5.3.3, the body of the main function does not need to be changed. Only the *checkTransitionAlgorithm* and the *preCheck* functions passed have to be adapted. Here below the function *equivalenceVerification* is displayed as a reminder.

```
def equivalenceVerification(graphA: List[GraphTransition], graphB:
    List[GraphTransition], idA: String, idB: String,
                            checkTransitionAlgorithm:
                                (List[GraphTransition],
                                List[GraphTransition], Set[(Int, Int)]) =>
                                Boolean,
                            preCheck: (List[(GraphTransition,
                                GraphTransition)], List[GraphTransition],
                                List[GraphTransition]) => Boolean):
                                Boolean = {

    val rootElementA = TransitionUtil.firstSource(graphA, Some(idA))
    val rootElementB = TransitionUtil.firstSource(graphB, Some(idB))

    val (initialStateSet, initialSet) = createInitialStateSet(graphA,
        graphB, Set(rootElementA), Set(rootElementB))

    if (!preCheck(initialSet, graphA, graphB))
      false
    else
      containRootElements(fixedPoint(graphA, graphB, initialStateSet,
          checkTransitionAlgorithm), rootElementA, rootElementB)
  }
```

This function inspects whether all states in *graphA* and *graphB* are part of H_O. To do so, *checkIfAllStatesPresentInInit* is called twice. One time for each graph.

```
  def preCheckBisimulation(initialSet: Set[(GraphTransition,
      GraphTransition)], graphA: Set[GraphTransition], graphB :
      Set[GraphTransition]): Boolean = {
    checkIfAllStatesPresentInInit(initialSet, graphA) &&
        checkIfAllStatesPresentInInit(initialSet, graphB)
  }
```

bisimulationCheck verifies if $(S_1, S_2) \in H_i$. Indeed, for all transitions of *graphA*,

the function tries to find an existing matching transition in *graphB*. If so, the function checks whether the tuples already belong to the previous set.

Then an extra step is added to match bisimulation definition. For all transitions of *graphB*, the function tries to find an existing matching transition in *graphA*. If so, the function checks whether the tuples already belong to the previous set.

```
def bisimulationCheck(nextTransitionA:Set[GraphTransition],
    nextTransitionsB:Set[GraphTransition], previousSet: Set[(Int,
    Int)]):Boolean =
  simulationCheck(nextTransitionA,nextTransitionsB,previousSet) &&
      simulationCheck(nextTransitionsB, nextTransitionA,
      previousSet.map(x=> (x._2,x._1)))
```

5.4 Weak bisimulation

5.4.1 Theoretical notions

Strong Bisimulation permits us to put apart processes with different deadlock behaviors along with many other desired properties that we have detailed in Section 5.1.

One might recall that we explained earlier in Chapter 2 that a process could afford τ transitions. Those transitions are internal to the process and unobservable from the outside world. The astute reader can remember, as well, that in Section 5.1 the last property studied was *observational equivalence*.

This last property is desirable to compare processes which are at a different level of abstraction. Indeed, if we want to verify if a specification is correctly implemented, we need to compare the CCS specification equation with the CCS implementation equation. As a consequence our notion of equivalence should abstract from internal transitions in process behaviors.

Let us consider the following specification and implementation equations for a chocolate vending machine :

$$\begin{cases} SCM = coin.\overline{chocolcate}.SCM \\ ICM = coin.\tau.\overline{chocolcate}.ICM \end{cases}$$

We expect those two systems to be equivalent because τ transition is not observable. However, the strong bisimulation expects each transition from one process to

be matched at least by another transition of the second process. This is impossible for the τ transition present in ICM and not in SCM.

To overcome this issue, our notion of bisimilarity has been adapted to abstract over τ actions. We define it as in [1, p. 57].

Let $T = (S, A, \longrightarrow)$ be a transition system. A binary relation $R \subseteq S$ x S is a *weak bisimulation* if whenever $(s, t) \in R$ and α is action (including τ):

if $s \xrightarrow{\alpha} s'$, then $t \stackrel{\alpha}{\Longrightarrow} t$' for some t such that s' R t';

if $t \xrightarrow{\alpha} t'$, then $s \stackrel{\alpha}{\Longrightarrow} s$' for some s such that s' R t'.

where we write $P \stackrel{\alpha}{\Longrightarrow} Q$ iff

$$\begin{cases} \text{either} & \alpha \neq \tau \text{ and there are processes } P' \text{ and } Q' \text{ such that } P(\xrightarrow{\tau})^* \ P' \xrightarrow{\alpha} Q'(\xrightarrow{\tau})^* \ Q \\ \text{or} & \alpha = \tau \text{ and } P(\xrightarrow{\tau})^* \ Q \end{cases}$$

Even though the notion of weak bisimilarity has proven itself to be useful to compare processes with τ actions, we leave it for a future work.

Nevertheless, we will give an overview on the changes needed in the bisimulation algorithm to process weak bisimilarity.

5.4.2 A weak bisimulation algorithm draft

The first step is the graph saturation. For each weak transition found in a graph, the algorithm transforms the corresponding original transition in a weak transition if this is not already the case.

As an example, our chocolate machine equations become:

$$\begin{cases} SCM = coin.\tau.\overline{chocolcate}.SCM \\ ICM = coin.\tau.\overline{chocolcate}.ICM \end{cases}$$

Then the strong bisimulation algorithm is applied on this new pair of saturated processes.

5.5 Conclusion

In this chapter we introduced a range of criteria to find a suitable notion of behavioral equivalence. Afterwards, we explored several possibilities: from the trace

equivalence to the weak bisimulation. We also provided an algorithm directly based on Tarski's fixed point theorem. Even though this algorithm was not the most efficient for computing (bi)simulation, it was really satisfying to see that a practical application could emerge from set theory. Despite being the most interesting notion tackled to compare processes, the weak bisimulation was not implemented in our tool, and was left for a future work.

During the writing of this chapter, the handouts on formal verification from Thierry Massart [13] and the chapters 3 to 4 from Reactive Systems [1] have proven themselves to be of great help.

6 Tutorial

The aim of this chapter is to give an informal introduction to the CCS workbench. All the features available will be briefly explained and illustrated. The software is composed of four different units: an editor to input CCS equations, a static view and a discovery view to display equations in graph form, and finally a unit to perform equivalence and preorder verifications.

6.1 CCS Editor

The CCS editor is used to input CCS equations. The CCS editor is available when one clicks on the link 'CCS Editor' in the main menu.

Process constant are in uppercase letters. Actions are in lowercase. Output action are prefixed with _. The syntax allowed in the editor is illustrated in Table 6.1:

CCS expression	Syntax
Nil Process	0
Process Definition	CLK =tick.CLK
Input Action / Prefixing	tick.tack.CLK
Output Action / Prefixing	coin._coffee.CM
Choice	coin.(_coffee.CM +_tea.CM)
Parallel composition	CM | CA
Restriction	(CM | CA)\{coin, coffee}

Table 6.1: CCS Editor Syntax

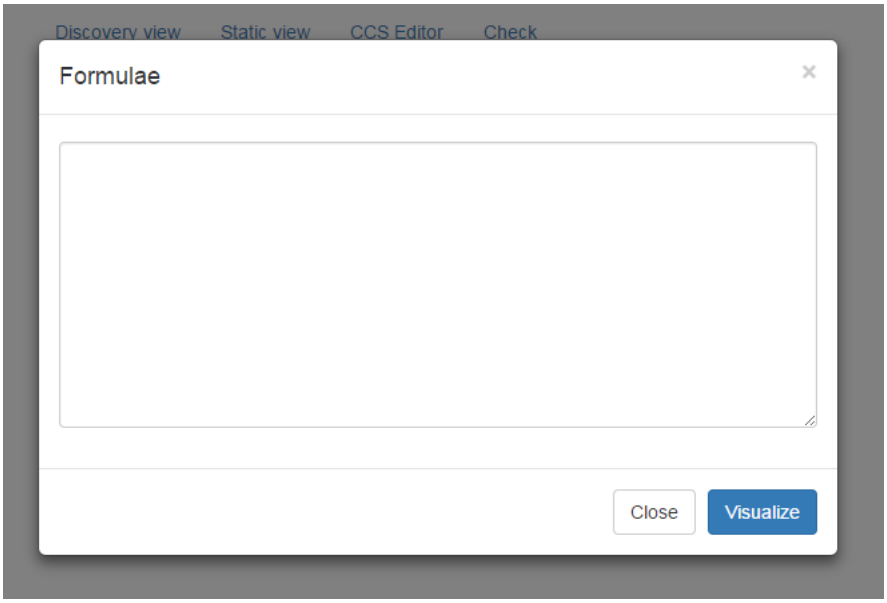


Figure 6.1: CCS Editor

An error message is displayed if the input is not correct. When the 'visualize' button is hit, the graphs associated with the formulae are available in discovery and static views.

6.2 Graph

6.2.1 Static View

The static view of the equation $A = a.b.c.\emptyset + l.a.\emptyset$ is illustrated in Figure 6.2. In case several equations are defined a selector allows the user to pick one.

6.2.2 Discovery View

The discovery view allows one to explore an equation step by step. In case several equations are defined a selector allows the user to pick one. To explore the graph the user has to choose one of the available nodes, and then click on it. As a result the next available transitions and nodes will be displayed.

The steps of the discovery view of the equation $A = a.b.c.\emptyset + l.a.\emptyset$ are illustrated in Figure 6.3.

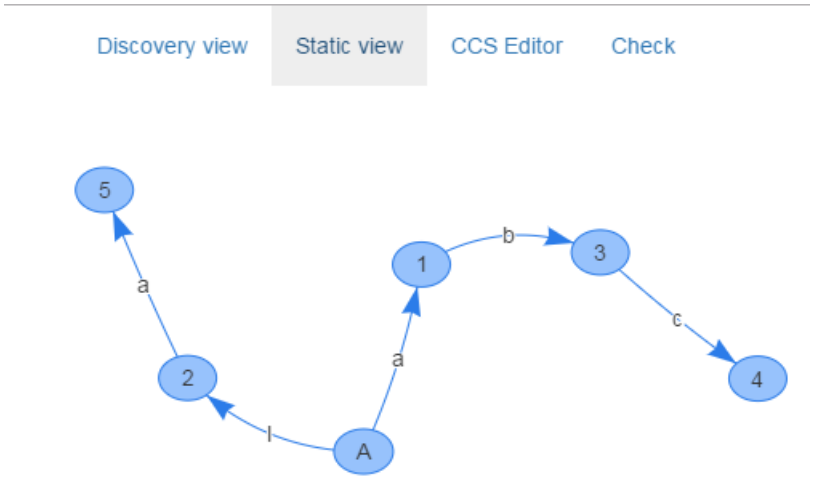


Figure 6.2: Static View

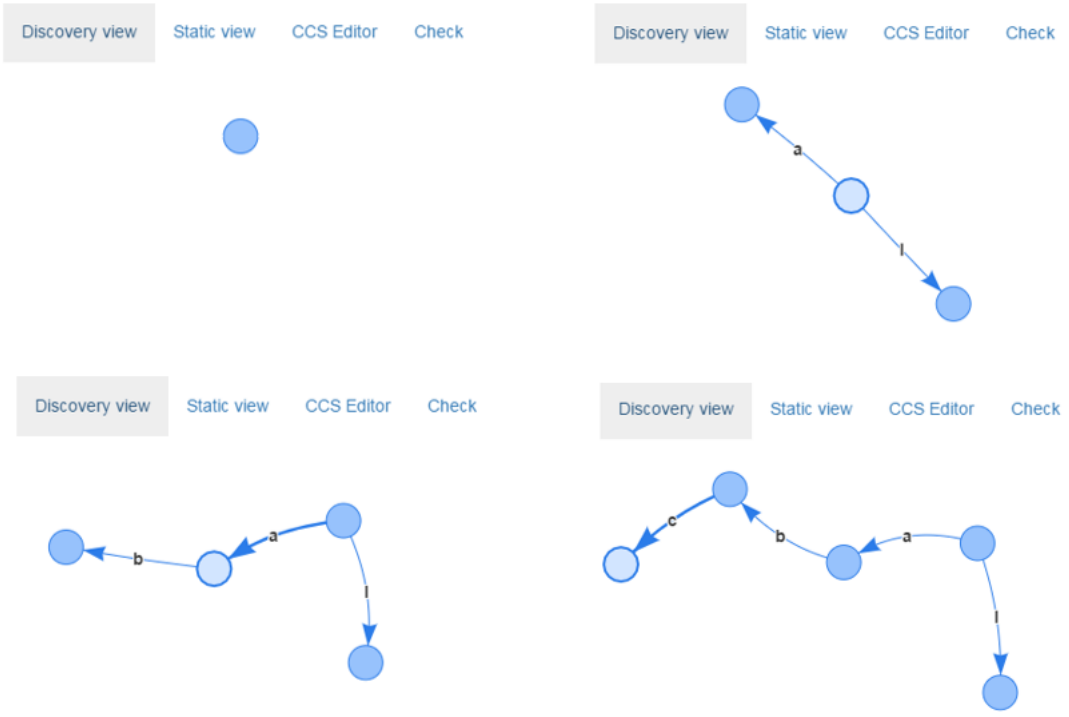


Figure 6.3: Discovery View

6.3 Equivalence Checking

Once at least two equations are defined, the user can verify equivalent behaviors by clicking on 'check'. Simulation, simulation equivalence, and strong bisimulation are performed on the selected formulae. A table with the results is displayed as portrayed in Figure 6.5. A green line means that the test was successful, whereas a red line means that it was unsuccessful.

Extra information can be obtained by clicking on a test result icon. The careful reader may notice that the three sets of tuples displayed in Figure 6.6 are actually the successive steps followed by the algorithm defined in Section 5.3.2 Strong Bisimulation. The same information is available for simulation equivalence and preorder.

The process definition is replaced by 0 in tuples. The left element of the tuple corresponds to a state from the first equation, whereas the right element corresponds to a state from the second equation. The last set contains the greatest (bi)simulation relation for the two equations.

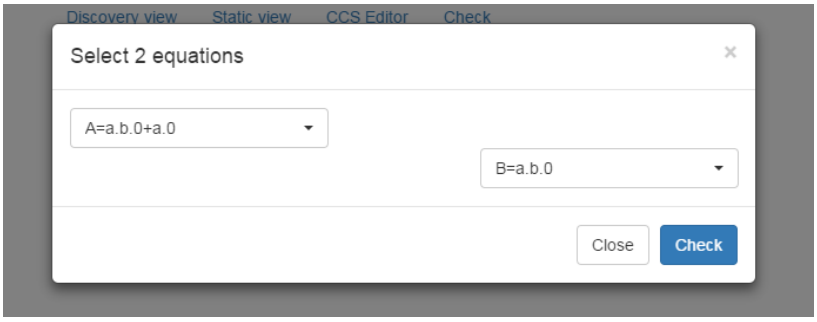


Figure 6.4: Verification - equation selection

Equivalence check

A Simulates B (simulation preorder)	
B Simulates A (simulation preorder)	
A and B are a bisimulation	
A and Bare a simulation equivalence	

Figure 6.5: Verification performed

Figure 6.6: Verification detailed mode

7 Conclusion

7.1 General Conclusion

This work was an opportunity for the two degree candidates to explore, learn and apply process theory. In doing so, we have developed and implemented a web based tool that allows one to specify, model and verify reactive systems through a range of features.

These features include graphic displays, simulation equivalence and preorder, and strong bisimulation equivalence. Displaying process algebra equations in the form of graphs facilitates the process of specifying, modeling and reasoning about systems. Comparing systems by simulation and strong bisimulation permits to proof that an implementation is equivalent to its specification. The process algebra is expressed in Milner's Calculus of Communicating Systems (CCS).

The originality of this work rest on a three-tier architecture : a back-end layer has the modeling language based on Scala, a front-end layer built on a web client based on JavaScript and vis.js, and a third layer with play as a web framework to link the front-end to the back-end Scala-based processing tool.

Back-end layer. Our implementation of the tool started by defining an abstract syntactic tree. The nodes and leaves have been defined with traits and classes. To extend their behavior, inheritance and mixin techniques have been applied. The syntax of Scala and its methods definition allowed us to create natural looking Domain Specific Language (DSL) for CCS.

To transform the flat text received into an abstract syntactic tree (AST), we used

the combinator parsing. Its implementation was straightforward, with a natural transposition of the CCS grammar. Such parsers are directly written in the host language which allowed us to use the expressiveness of the host language. Therefore, pattern matching, high-order functions, parametric polymorphism and the previously defined CCS DSL were immediately available.

During subsequent steps, transitions generation and equivalence verifying functions were implemented with much benefit from the complementary strengths of object oriented and functional paradigms. A major obstacle was the steep learning curve to become reasonably proficient with the diversity of concepts available in the Scala language.

Front-end layer. *vis.js* enabled us to display (LTS) transitions as graphs. Fortunately, the framework is well documented and its usage was straightforward. However, we were limited in our usage of it, and thus we have not tested it with a large amount of dynamic datasets. Another limitation was that the manipulation and interactions of the graphs were limited by the basic nature of the workbench.

Play. The Play web framework allowed us to transfer back-end Scala transition objects to the view by a direct JSON transformation. Type-safe compiled templates were used to build the HTML view with the Scala objects. The integration of Play with Scala was seamless, with convenient developer friendly features such as the hot-code reloading, hit-refresh work flow, and error handling. The integration drawbacks of Play were not relevant to our case.

In final conclusion, the work to produce this thesis shows an effective stack of technologies for building a web based workbench for CCS. Even if the prototype we developed did not support all the features of CCS, the work still demonstrated the complementary strengths of oriented-object and functional programming provided by Scala.

In many fields of endeavor, one often hears the expression “see one, do one, teach one”, indicating that the most profound understanding of a subject is achieved only when one can impart it effectively to someone else. In our case, that “else” was a computer that we taught to perform algorithms.

7.2 Limitations and future work

This section points out the limitations of the workbench and outlines directions for future work.

Limitations. As mentioned above and detailed in Section 4.7.1, our prototype tool did not cover unguarded CCS equations. To try to overcome this shortfall, a rewriting unit has been added to the tool. This unit replaces unguarded process reference by their definitions, but it still does not handle complex cases.

Another limitation was that, for the sake of simplicity, the | operator only covers the basic case with two operands, such as : $A = C \mid D$ where C and D are defined as terminal CCS equations.

Managing all cases in the current prototype would be tedious due to its chosen design. A better solution will require future work.

Related and Future work. To better grasp the re-factoring needed, let us give the details of the approach tailored by CAAL. CAAL is a modern web based tool written in TypeScript enabling graph display and verification of reactive systems.

During the parsing stage for each encountered process, transitions are produced on the fly. To do this, it requires the process to be sliced into sub-process/sub-states as in: $a.b.\emptyset \longrightarrow b.\emptyset \longrightarrow \emptyset$.

The process preserves a reference to its sub-process(es). The union of the transitions of each sub-process allows to acquire the whole set of the process transitions. The set of transitions for each process/sub-process are maintained in a map with the corresponding ID.

By having a map with all sub-process transitions and a link between processes and sub-processes, we could simplified our software. In this manner, the transitions of reference to a process in a formula can be treated as a straightforward call to their identifiers in the map. Cases such as $A + B$, $A \mid B$, and $a.b.c(A + (A \mid B))$ would be treated effortlessly.

We think that such an approach could permit to cover all the cases of the CCS language without limitations or extra complexity.

In addition, a tool of formal verification needs to be specified and proven correct. In a future work we would seek to benefit from the referential transparency, given by most of the functions written inside the tool, to prove its correctness.

Bibliography

[1] Aceto, L., Ingólfsdóttir, A., Larsen, K. G., and Srba, J. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, New York, NY, USA, 2007.

[2] Andersen, J. R., Andersen, N., Enevoldsen, S., Hansen, M. M., Larsen, K. G., Olesen, S. R., Srba, J., and Wortmann, J. K. *CAAL: Concurrency Workbench, Aalborg Edition.* Springer International Publishing, Cham, 2015, pp. 573–582.

[3] Bort, G. The Play Framework. `https://www.playframework.com/`, 2017. Accessed: 2017-04-01.

[4] Cambridge University Press. Reactive Systems: Modelling, Specification and Verification: Tools. `http://rsbook.cs.aau.dk/?page_id=34`, n.d. Accessed: 2017-05-02.

[5] Documentation, S. Case Classes. `http://docs.scala-lang.org/tutorials/tour/case-classes.html`, n.d. Accessed: 2017-04-23.

[6] Fernandez, J.-C. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming 13*, 2 (1990), 219 – 236.

[7] Guillaume, B. Play Framework Presentation. `https://fr.slideshare.net/GenevaJUG/play-framework-presentation`, 2010. Accessed: 2017-04-17.

[8] Haller, P., and Odersky, M. Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science 410*, 2-3 (2009), 202–220.

[9] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.

[10] Jonas, B. Why did Typesafe Select Play for their Stack instead of Lift. `https://www.quora.com/Why-did-Typesafe-select-Play-for-their-stack-instead-of-Lift/answer/Jonas-Bon%C3%A9r?srid=vjGq`, 2013. Accessed: 2017-04-17.

[11] Kučera, A., and Mayr, R. Simulation Preorder over Simple Process Algebras. *Information and Computation 173*, 2 (2002), 184 – 198.

[12] Labun, E. Combinator Parsing In Scala. Tech. rep., Technische Hochschule Mittelhessen, 2012.

[13] Massart, T. Formal Verification of Computer Systems. `http://www.ulb.ac.be/di/verif/tmassart/Verif/syllabus_verification_2p.pdf`, 2013. Handout - 2013 - Accessed: 2017-04-10.

[14] Matiello, Pedroand de Melo, A. C. V. *PiStache: Implementing π-Calculus in Scala.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 76–91.

[15] Milner, R. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[16] Milner, R. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[17] Odersky, M., and al. An Overview of the Scala Programming Language. Tech. rep., EPFL Lausanne, Switzerland, 2006.

[18] Odersky, M., and al. Scala Language Specification. `https://www.scala-lang.org/files/archive/spec/2.11//`, 2014. Accessed: 2017-05-20.

[19] Odersky, M., and al. Scala: Evaluation Strategies and Termination. `http://lamp.epfl.ch/files/content/sites/lamp/files/teaching/progfun/slides/week1-3-no-annot.pdf`, 2016. Accessed: 2017-04-08.

[20] Sagar, G. Embedded Software Market Trends & Forecast from 2016 to 2023. https://www.linkedin.com/pulse/embedded-software-market-trends-forecast-from-2016-2023-gavhane-sagar, 2016. Accessed: 2017-05-20.

[21] Seth, A., Singla, A. R., and Aggarwal, H. *Service Oriented Architecture Adoption Trends: A Critical Survey", bookTitle="Contemporary Computing: 5th International Conference, IC3 2012, Noida, India, August 6-8, 2012. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 164–175.

[22] Tarski, A. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math. 5*, 2 (1955), 285–309.

[23] Yevgeniy-Brikman. What are the Pros and Cons of Play Framework 2 for a Scala Developer. https://www.quora.com/What-are-the-pros-and-cons-of-Play-Framework-2-for-a-Scala-developer/answer/Yevgeniy-Brikman, 2013. Accessed: 2017-04-17.

[24] Zenger, M., and Odersky, M. Independently Extensible Solutions to the Expression Problem. Tech. rep., EPFL Lausanne, Switzerland, 2004.